VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PODPORA JAVASCRIPTU V ZOBRAZOVACÍM STROJI HTML

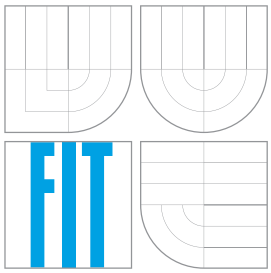
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADIM LOSKOT

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PODPORA JAVASCRIPTU V ZOBRAZOVACÍM STROJI HTML

JAVASCRIPT SUPPORT IN AN HTML RENDERING ENGINE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADIM LOSKOT

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK BURGET, Ph.D.

BRNO 2014

Abstrakt

Tato diplomová práce vznikla z důvodu vytvoření rozšíření pro experimentální zobrazovací stroj CSSBox o podporu skriptování v jazyce JavaScript. V teoretické části práce popisuje architekturu zobrazovacího stroje a uvádí do problematiky skriptování v HTML dokumentech podle doporučené specifikace HTML 5. Práce se zabývá důkladnou analýzou existujících skriptovacích strojů a jejich rozhraní, kterých by bylo možné využít pro implementaci rozšíření. V závislosti na teoretických znalostech se práce zaměřuje na skriptovací stroj Rhino a představuje abstraktní návrh jeho zakomponování do projektu CSSBox. Závěrem hodnotí kompatibilitu a výkonnost implementovaných funkcionalit a zamýšlí se nad možnostmi jejich dalšího budoucího rozšíření.

Abstract

This Master's thesis was written to create the extension for an experimental rendering engine CSSBox about scripting support in JavaScript language. In the theoretical section the thesis describes the architecture of the rendering engine and introduces problems of the scripting in HTML documents according to the recommended HTML 5 specification. This thesis deals with thorough analysis of existing scripting engines and their interfaces, which could be used for the extension implementation. Depending on the knowledge gained from the previous parts this thesis focuses only on the scripting engine Rhino and introduces an abstract design of its integration into the CSSBox project. At the end it evaluates the reliability and efficiency of the implemented functionalities and considers possibilities of their further development.

Klíčová slova

JavaScript, stroj, Java, HTML, CSSBox, Rhino, webový prohlížeč, skriptovací Java API

Keywords

JavaScript, engine, Java, HTML, CSSBox, Rhino, web browser, Java Scripting API

Citace

Radim Loskot: Podpora JavaScriptu v zobrazovacím stroji HTML, diplomová práce, Brno, FIT VUT v Brně, 2014

Podpora JavaScriptu v zobrazovacím stroji HTML

Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně a uvedl všechny literární prameny a publikace, ze kterých jsem čerpal, pod vedením pana Ing. Radka Burgeta, Ph.D.

.....
Radim Loskot
14. ledna 2014

Obsah

Kapitola 1

Úvod

Internetové prohlížeče umožňují rychle a pohodlně procházet webový obsah a tzv. HTML dokumenty. HTML dokumenty nemusí být výhradně umístěny pouze na webu a na vzdálených úložištích, ale mohou přicházet i v rámci elektronické pošty nebo být součástí aplikační nápovědy aj. Pro zobrazování HTML dokumentů používáme HTML zobrazovacích strojů. Jedním takovým strojem je CSSBox.

CSSBox je projekt experimentálního (X)HTML/CSS zobrazovacího stroje napsaného v čistém jazyce Java. Projekt se snaží vyplnit prostor v implementacích zobrazovacích strojů v Javě, jejichž vývoj většinou ustal. Hlavním cílem projektu je poskytnout úplné a dále zpracovatelné informace o rozložení dokumentu. Vyjma analyzování dokumentu umožnuje již v samotném základu vizualizovat dokument jako prostý obrázek. V rámci samostatného podprojektu SwingBox dále díky komponentě `BrowserPane` přidává pokročilé grafické rozhraní pro zobrazování dokumentu, které dokáže reagovat i na události od uživatele.

V současné době CSSBox implementuje (X)HTML/CSS 2.1 analyzátor a výše zmíněné zobrazovací komponenty. Cílem této práce bylo projekt CSSBox rozšířit o možnost skriptování v dokumentu a implementovat vybranou základní podmnožinu skriptovacího jazyka JavaScript. Jakým způsobem bylo integrace docíleno, je popsáno v následující práci.

V úvodu této práce – kapitole 2.1 se pojednává o způsobech, jakými mohou být vloženy do HTML dokumentů skripty, a je demonstrován referenční přístup pro přidání skriptování do dokumentů podle doporučené specifikace HTML 5. Základní architektura použitých závislostí – projektů CSSBox a SwingBox je uvedena v kapitolách 2.2 a 2.3.

Kapitola 3 je zaměřena na analýzu dostupných skriptovacích strojů v současnosti a rozbor jejich API. Kapitola blíže popisuje skriptovací engine Rhino, který byl využit pro implementaci JavaScriptového enginu. V rámci této kapitoly je představeno i standardní skriptovací Java API, jež lze pro skriptování v Javě využít.

V kapitole návrhu – kapitole 4 je nastíněn základní abstraktní návrh celé integrace nového rozšíření do projektu SwingBox. Postupně jsou ukázány jednotlivé náležitosti, které bylo zapotřebí vykonat, aby bylo možné integraci provést. V závěru kapitoly 4.5 je demonstrována „registrace“ samotného rozšíření do projektu SwingBox.

Kapitola implementace – kapitola 5 popisuje již implementované řešení, vytvořené jádro uživatelského agenta a jeho podstatu pro zbylá implementovaná rozhraní (kapitola 5.1). Kapitola též představuje způsob, jakým lze vytvářet konkrétní klientské skriptovací enginy (kapitola 5.2), a zároveň uvádí implementovaný engine JavaScriptu (kapitola 5.3). Poslední podkapitola 5.5 znázorňuje dvě ukázkové aplikace, které sloužily pro předvedení hotové funkčnosti a k otestování implementované podmnožiny JavaScriptu.

Testováním, rozborem a případným rozšířením implementovaného řešení se podrobně zaobírá kapitola 6. V podkapitole 6.1 jsou uvedeny techniky, jakými probíhalo testování a ověření správné funkčnosti. Jelikož bylo žádoucí porovnat implementované řešení s ostatními internetovými prohlížeči, hodnotí kapitola 6.2 implementované řešení z hlediska jeho výkonnosti. Kapitola 6.3 dále vytváří souhrn veškeré implementované funkčnosti a klasifikuje kompatibilitu implementovaného řešení. Závěrem jsou v kapitole 6.4 diskutovány některé případné cesty budoucí vývoje.

Kapitola 2

Analýza použitých technik

V rámci této práce se budeme zabývat integrací podpory skriptování v JavaScriptu pro stroj CSSBox. Abychom mohli provést návrh knihovny implementující rozšíření stroje CSSBox, je zapotřebí upřesnit, jak jsou skripty v HTML dokumentech reprezentovány, jak jsou vykonávány a do jakého rozhraní je budeme integrovat.

Problematikou skriptování v HTML dokumentech se zabývá kapitola 2.1. Na projekt CSSBox a jeho architekturu je zaměřena kapitola 2.2. Komponenta, která bude použita pro zobrazování HTML dokumentů a která bude implementovat rozšíření skriptování v těchto dokumentech, je popsána v kapitole 2.3.

2.1 Klientské skripty v HTML dokumentu

Klientský skript je program, který může doprovázet HTML dokument nebo být do něj přímo vložen. Skript je vykonáván na straně klienta buď ihned po jeho načtení, po načtení celého dokumentu nebo při výskytu některé události. Skripty mohou:

- sloužit k modifikaci obsahu dokumentu,
- být spuštěny událostmi ovládacích prvků a jinými událostmi prohlížeče,
- být využity pro validaci a odesílání obsahu formuláře na server.

Pro vložení nového skriptu do HTML dokumentu a jeho případné vykonání máme několik způsobů, např.:

1. Uvést skript v dokumentu ve speciální značce `<script>`;
2. Vložit skript do atributu elementu určeného pro událostní obslužnou rutinu, např. do atributu `onclick`. Skript se bude vykonávat vždy po výskytu dané události;
3. Vložit skript do dokumentu dynamicky tak, že vložíme do dokumentu pomocí skriptu nový element `<script>` a definujeme jeho vykonávací kód.

Skripty ve značkách `<script>` jsou spouštěny samostatně, vždy pouze jednou a nikoliv opakovaně jako skripty vytvářené v atributu pro událostní obslužné rutiny. Během zpracovávání dokumentu jsou jednotlivé skripty v těch značkách vykonány. Jak a v jakém pořadí spouštění skriptů probíhá, popíšeme dále v této kapitole.

V rámci samotné značky `<script>` můžeme podle specifikace HTML 5 [19] rozlišit celkem 6 atributů. Všechny atributy řídí vykonávání skriptu v určitém momentu jeho zpra-

covávání a jejich dynamická změna nemá žádný efekt na průběh skriptu. Specifikace uvádí následující atributy:

1. `src` – URL[1] adresa souboru s externím skriptem;
2. `type` – MIME[2] typ skriptu;
3. `charset` – typ kódování souboru s externím skriptem;
4. `async` – udává, zda bude skript spuštěn asynchronně;
5. `defer` – specifikuje odložení spuštění skriptu až do dokončení načtení dokumentu;
6. `crossorigin` – pokud je atribut přítomem, bude na server zaslán CORS[3] požadavek.

Kromě samotných atributů elementu skriptu specifikace HTML 5 doporučuje asociovat s elementem `<script>` i speciální příznaky. Příznaky jsou nastavovány na základě toho, kde byl element skriptu zpracován a na základě výše zmíněných atributů. Uvedenými příznaky ve specifikaci jsou:

1. `already-started` – značí, zda byl skript již vykonán. Při klonování elementu `<script>` se musí příznak sdílet;
2. `parser-inserted` – příznak je nastaven pro všechny skripty vložené do dokumentu HTML nebo XML parserem během načítání dokumentu nebo při vložení pomocí `document.write()`. Po vytvoření elementu `<script>` není tento příznak nastaven;
3. `force-async` – během počátečního zpracování dokumentu je příznak nastaven. Skripty nemají tento příznak nastaven, pokud jsou vytvořeny a vloženy do dokumentu dodatečně nebo pokud mají specifikovaný atribut `async`;
4. `ready-to-be-parser-executed` – příznak používaný pouze pro skripty, které byly vloženy dodatečně některým z parserů dokumentu. Příznak je nastaven, pokud dojde k úspěšnému načtení skriptu z externího zdroje;
5. `script-type a script-charset` – jsou příznaky, které pouze reflektují odpovídající atributy elementu `<script>`. Pokud nejsou atributy nastaveny, nebo jsou nastaveny špatně, pak obsahují tyto příznaky opravené nebo výchozí hodnoty těchto atributů.

Nastavování části výše zmíněných příznaků by měl obsluhovat parser dokumentu během tvorby elementu `<script>`. Před spuštěním skriptu by se podle specifikace HTML 5 měla provést příprava skriptu. Příprava zahrnuje několik kroků, které nastaví příznaky např. `script-type` a `script-charset`, případně upraví příznaky nastavené parserem.

Příprava elementu `<script>` pro jeho budoucí spuštění je provedena po dokončení jeho parsování nebo pokud skript nemá nastaven příznak `parser-inserted` a pokud nastane:

1. vložení elementu požadovaného skriptu do dokumentu,
2. vložení nového textového uzlu nebo fragmentu dokumentu do již existujícího skriptu,
3. nastavení atributu `src`, který nebyl předtím nastaven.

[1] URL (Uniform Resource Locator) – řetězec sloužící k jednoznačné identifikaci zdroje

[2] MIME (Multipurpose Internet Mail Extensions) – standard definující typy internetového média

[3] CORS (Cross-origin resource sharing) – Mechanismus umožňující stahování zdrojů z více povolených domén než pouze z domény, ze které pochází zdroj.

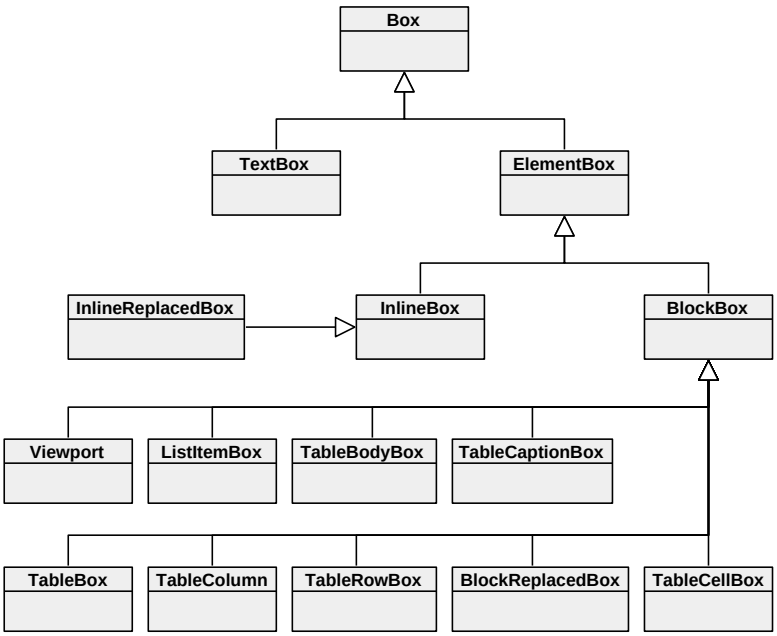
Prvotní fáze přípravy skriptu, jíž výsledkem je nastavení příznaku `already-started`, tj. spuštění skriptu, je znázorněna v příloze B. Během přípravy dochází také k zahájení načítání skriptů z externích zdrojů a umísťování jejich spuštění do front skriptů ke spuštění, aniž by docházelo k čekání, než jsou skripty kompletně načteny.

Pokud spouštíme skript existující přímo v dokumentu, je jeho vykonávání zahájeno ihned a to i bez ohledu na to, zda předchozí skripty dokončily své provádění. Všechny skripty, které mají uvedený externí zdroj, nejsou `parser-inserted` a byly již načteny, jsou spouštěny také okamžitě bez zbytečného čekání. Naopak problematika umísťování blokujících skriptů do front skriptů ke spuštění z důvodu čekání na zpracování stylů nebo `parser-inserted` skriptů je značně obsáhlá. Více o zpracovávání těchto skriptů si lze dočíst v samotné specifikaci HTML 5 [19].

2.2 Renderovací stroj CSSBox

CSSBox je projekt (X)HTML/CSS renderovacího stroje napsaného v čistém jazyce Java. Projekt klade velký důraz na poskytnutí informací o zpracovávané stránce.

Renderovací stroj CSSBox očekává na svém vstupu model dokumentu. Dokument je v rámci stroje získáván pomocí parseru NekoHTML, ale může být získáván i jiným způsobem. Rozhraním pro všechny zdroje dokumentu je abstraktní třída `DOMSource`. Pro zpracování stylů dokumentu slouží třída `DOMAnalyzer`, která převede aktuální dokument na rozložení stránky. Výsledný model rozložení stránky se využívá pro zobrazení stránky. V knihovně je pro tento účel implementována komponenta `BrowserCanvas`.



Obrázek 2.1: Hierarchie zobrazovacích komponent třídy `Box`

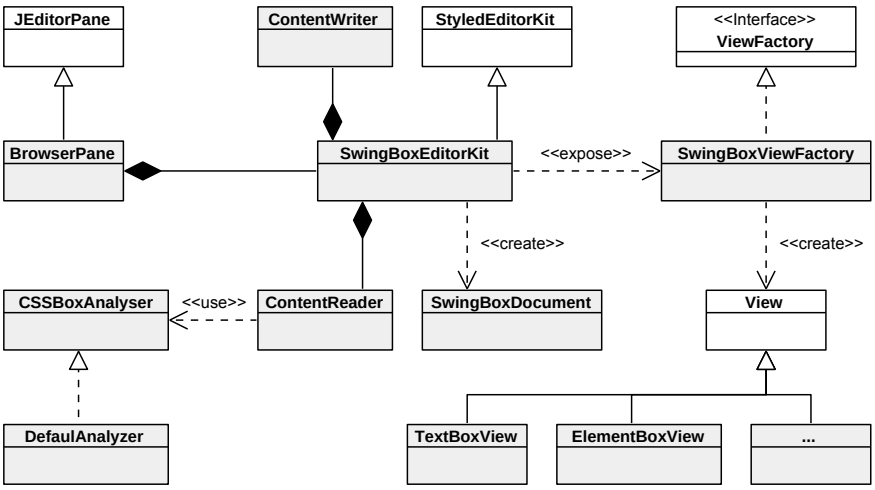
Komponenta `BrowserCanvas` vytváří strom vykreslitelných objektů třídy `Box`. Strom je vytvářen ze vstupního dokumentu a výstupu analyzátoru stylů. Každý vykreslitelný prvek HTML stránky se pojí se specializovanou třídou `Box` tak, jak je znázorněno na obrázku 2.1.

Typ boxu nemusí být určen pouze typem elementu dokumentu, ale i jeho styly. Pokud má například blokový prvek nastaveno zobrazení na řádku, tak bude ve stromu reprezentován třídou `InlineBox` a nikoliv třídou `BlockBox`.

2.3 Projekt SwingBox

V rámci podprojektu SwingBox, který je součástí projektu CSSBox, byla vyvinuta komponenta `BrowserPane` [26], která rozšiřuje funkčnost základního zobrazování dokumentu komponentou `BrowserCanvas`. Komponenta `BrowserPane` vychází ze třídy `JEditorPane`, ve které předefinovává výchozí zobrazování (X)HTML dokumentů. Pro zobrazování těchto dokumentů využívá třídu `SwingBoxEditorKit`, která zařizuje veškerou logiku zobrazování.

Komponenta `BrowserPane` umožňuje načítat dokumenty předáním jejich URL adresy, předáním dokumentu jako textový řetězec nebo předáním vstupního streamu s dokumentem. Na základě typu dokumentu, který má být zobrazen, komponenta dále volá pro čtení dokumentu odpovídající kit – v našem případě `SwingBoxEditorKit`. Kit zařizuje s použitím třídy `ContentReader` načtení rozvržení stránky (viz kapitola 2.2) a jeho převod na reprezentaci dokumentu `SwingBoxDocument` tak, jak ho přijímá rozhraní komponenty `JEditorPane`. Třída `ContentReader` provádí převod rozvržení stránky na prvky dokumentu, tzn., převádí objekty třídy `Box` na objekty třídy `ElementSpec`. Veškeré načítání dokumentu ve formě DOM[4] a jeho rozvržení třídou `DOMAnalyzer` je prováděno ve třídě `CSSBoxAnalyzer`, na kterou je reference uvnitř třídy `ContentReader`.



Obrázek 2.2: Diagram tříd komponenty `BrowserPane`

Prvky dokumentu `SwingBoxDocument`, které vytváří třída `ContentReader`, jsou vykreslovány na zobrazovací plochu `BrowserPane` ve formě pohledů `View`. Vykreslování řídí automaticky komponenta `JEditorPane`, která zná referenci na továrnu pohledů `SwingBoxViewFactory`. Továrna pohledů je používaná komponentou `JEditorPane` vždy při potřebě vykreslit požadovaný dokument. Továrna se využívá pro vytváření pohledů pro každý prvek dokumentu `SwingBoxDocument`. Jelikož prvky dokumentu `ElementSpec` v sobě zapouzdřují informaci o tom, jaký box obalují, je možno vytvořit pohledy, které přesně korespondují s množinou boxů uvedených v kapitole 2.2.

[4]DOM (Data Object Model) – objektově orientovaná reprezentace XML nebo HTML dokumentu

Kapitola 3

Analýza podpory skriptování v jazyce JavaScript v Javě

Tato kapitola se zabývá analýzou skriptování a nástrojů, které lze využít pro tvorbu skriptů. Zaměřuje se na skriptovací jazyk JavaScript a jeho použití v Java aplikacích. Kapitola záměrně popisuje zejména skriptovací stroj Rhino a standardní skriptovací Java API, jež lze pro skriptování použít.

Skriptovací jazyky jsou programovací jazyky, které nám umožňují psát tzv. skripty. Na rozdíl od kompilovatelných zdrojových kódů, je kód skriptu vyhodnocován a interpretován za běhu skriptu. Většina skriptovacích jazyků je typována dynamicky za běhu skriptu, což nám umožnuje vytvářet proměnné bez specifikace typu. Z důvodu dynamické typové kontroly můžeme ve skriptovacích jazycích použít jednu proměnnou pro uložení více typů. Skriptovací jazyky mají jednoduchou syntaxi a umožňují v určitých případech vyřešit poměrně složité problémy pomocí relativně krátkého kódu, efektivně a poměrně v krátkém čase.

Cílem této diplomové práce je umožnit provádění skriptů z kódu Javy. V Javě bylo implementováno mnoho skriptovacích jazyků využívajících různá rozhraní pro skriptování, což bylo mj. i důvodem pro vznik obecného skriptovacího Java API (viz kapitola 3.3). Mezi nejrozšířenější skriptovací jazyky Javy můžeme zařadit BeanShell, s velmi podobnou syntaxí jakou disponuje Java, a další implementace již existujících jazyků v Javě, jako je např. Jython – implementace Pythonu, JRuby – implementace Ruby, Groovy aj.

Interpretů JavaScriptu implementovaných v Javě běžících na JVM[1] není mnoho. Z důvodu výkonnosti je většina interpretů JavaScriptu pro webové prohlížeče napsána nativně v C nebo C++ a využívá JIT[2] techniky. Jmenovitě nejznámější nativní JavaSriptová jádra jsou z rodiny „Monkey“, kterou spravuje společnost Mozilla, a která jsou postupně využívána ve webovém prohlížeči Firefox. Dalšími jádry ostatních webových prohlížečů je V8 použité v Google Chrome, Carakan v Opeře, Chakra v Iternet Exploreru nebo JavaScriptCore v prohlížeči Safari.

Nejpoužívanější interprety JavaScriptu cílené pro Java aplikace máme v současnosti pouze tři. Nejznámějším interpretem je již zmíněné Rhino (kapitola 3.1) [14], které je i dodáváno v Oracle implementaci JDK[3] 6 a 7, kde realizuje výchozí skriptovací stroj pro Javu. Dalším interpretem s velkým budoucím potenciálem, který je součástí nedávno vydané

[1] JVM (Java Virtual Machine) – virtuální stroj Javy zpracovávající mezikód jazyka Java (Java bytecode)

[2] JIT (Just in Time) – metoda překladu urychlující běh skriptu překladem kódu do strojového jazyka

[3] JDK (Java Development Kit) – balík základních nástrojů potřebných pro vývoj aplikací pro platformu Java

Oracle implementace JDK 8, je interpret Nashorn [13] (kapitola 3.2). Oba zmíněné interprety JavaScriptu implementují standardní skriptovací Java API. Rhino bylo o skriptovací API rozšířeno firmou Sun, kdežto Nashorn je vyvíjen s tímto API zprvu počátku. Posledním a nepříliš používaným interpretem je YAJI [25], který se snaží oživit interpret FESI [5] tím, že do něho přidává většinu vlastností z nejnovější specifikace ECMAScript.

3.1 Rhino

Rhino je implementace JavaSriptového jádra napsaná v programovacím jazyce Java. Projekt Rhina byl zahájen v roce 1997 firmou Netscape, kdy firma potřebovala integrovat podporu JavaScriptu do budoucího prohlížeče založeném na platformě Java [16]. Rhino vzniklo portem nativní knihovny SpiderMonkey, takže i nyní si nelze v kódu nevšimnout Java ekvivalentů nepodmíněných skoků [16].

Projekt prohlížeče, tzv. Javagatoru, byl ovšem brzy pozastaven a Rhino zůstalo dále jen velmi pozvolna vyvíjené několika vlastníky license, včetně společnosti Sun [16]. V roce 1998 bylo Rhino uvolněno společnosti Mozilla a držitelé licencí se dohodli, že vydají Rhino jako svobodný software [16]. Nyní je Rhino spravováno společností Mozilla.

3.1.1 Vlastnosti Rhina

V současnosti je vydáno Rhino s označením 1.7R4 [11, 12]. Nejnovější verze implementuje všechny vlastnosti Javasriptu 1.7 – je plně ekvivalentní s 3. vydáním standardu ECMA-262 ECMAScript. Nejnovější verze také přidává některé nové vlastnosti, jako jsou doplňky pro práci s poli a podpora E4X[4]. Od verze 1.7R3 Rhino přidává i částečnou podporu Javasriptu 1.8 a 5. vydání ECMAScriptu, která byla nejnovější verzí 1.7R4 zejména optimalizována a jen mírně rozšířena.

Významnou předností Rhina je jeho dobrá provázanost s Javou. Napojení na jazyk Java a umožnění hostování objektů Javy nám zpřístupňuje z kódu JavaScriptu téměř libovolný balíček a třídu z Javy. Výjimky pro třídy, které by neměly být zpřístupněny, jsou definované bezpečnostními politikami ve třídě `SecurityManager` (Java API) a dodatečně třídou `ClassShutter` (knihovna Rhino). Jamile je přístup do třídy povolen, lze z kódu JavaScriptu tuto třídu konstruovat, volat její libovolnou metodu či přistupovat k jejím atributům. Aby bylo možné objekt Javy v JavaScriptu používat, musí vždy implementovat rozhraní `Scriptable` (kapitola 3.1.2). V případě hostování nativního objektu Javy, je rozhraní `Scriptable` implementováno automaticky pomocí speciální zapouzdřovací továrny, jež je instancí třídy `WrapFactory`.

Mezi další přednosti Rhina patří možnost rozdělení skriptů do modulů tak, jak bylo specifikováno skupinou CommonJS[5], a možnost vysledovat zdroj, odkud skript pochází a zpracovávat skript podle odpovídající bezpečnostní politiky. Bezpečnostní politiky v Rhinu vycházejí z Netscape Navigatoru a jejich aplikace je založená na kontrole URL zdroje [17].

Rhino umožňuje dva režimy – interaktivní a kompilovaný. V kompilovaném módu je kód přeložen do mezikódu Javy a interpretován dále v JVM. V interaktivním módu je skript spouštěn jednoduchým interpretem, aniž by docházelo ke generování mezikódu Javy.

[4] Rozšíření programovacího jazyka ECMAScript, které přidává podporu nativního XML a vychází z normy ECMA-357 ECMAScript pro XML.

[5] CommonJS – Skupina lidí snažící se vytvořit jednotné JavaScriptového prostředí pro servery, desktopy a prohlížeče.

Poslední zajímavou vlastností knihovny Rhino je poskytování z kódu JavaScriptu podpory pro implementaci rozhraní a abstraktních tříd Javy. Implementaci nedefinovaných metod z JavaScriptu zajišťuje třída objektového adaptéru `JavaAdapter`. Adaptér vytváří pomocí reflexe objekty Javy a implementuje do nich navíc metody rozhraní `Scriptable`, čímž zpřístupňuje tyto objekty pro přímé použití ve skriptech JavaScriptu. Adaptér můžeme buď přímo zavolat nebo ho nechat knihovnou Rhino automaticky odvodit.

3.1.2 Skriptování s Rhinem

Zdrojové kódy Rhina jsou strukturovány do několika hlavních balíků. Pro konstrukci skriptů v Javě využijeme hlavně veřejné API z balíku `org.mozilla.JavaScript`.

Základním prvkem pro spuštění jakéhokoliv skriptu je objekt třídy `Context`, jenž nese vláknově specifické informace o prostředí pro běh skriptů. Každé programové vlákno, které vyžaduje spuštění skriptu, by si mělo asociovat vlastní kontext zavoláním `Context.enter()` a uvolnit zavoláním `Context.exit()` [18].

Úložiště pro objekty nejvyšší úrovně nazýváme v JavaScriptu jako tzv. scope. Do scope vkládáme všechny vlastní objekty. Scope představuje jakousi množinu objektů. Objekt scope je v Rhinu klasický JavaScriptový objekt, který implementuje třídu `Scriptable` a jenž by měl obsahovat základní stadardní objekty `Object` a `Function`. Inicializovaný scope můžeme získat zavoláním metody `initStandardObjects()` objektu kontextu. Důležitou vlastností scope je, že je kontextově nezávislý, i přestože scope může být vytvořen z kontextu [18]. Scope vytvořený jedním kontextem můžeme vyhodnotit s využitím jiného kontextu. Ve vícevláknových aplikacích lze nechat jeden scope vyhodnotit více odlišnými kontexty zároveň. Rhino zajišťuje, že přístup k vlastnostem objektu je atomický.

Objekty Javy, které chceme zpřístupnit do kódu skriptu, by měly implementovat rozhraní `Scriptable` a poskytnout metody pro práci s vlastnostmi objektu: `get()`, `put()`, `has()` a `delete()`. Základní implementaci rozhraní `Scriptable` a přístup k atributům objektu pomocí hashovací tabulky poskytuje v Rhinu třída `ScriptableObject`. Kromě toho třída `ScriptableObject` implementuje některé důležité metody pro definování hostovaných objektů z balíků Javy, např. statickou metodu `defineClass()`.

Posledním důležitým rozhraní je rozhraním `Function`. které je implementováno všemi funkčními objekty Javasriptu. Pokud chceme funkci zavolat, můžeme k tomu použít metody `call()` nebo `construct()` v závislosti na tom, jestli funkce má být volání s klíčovým slovem `new`.

JavaScript kromě základního objektu `Object`, který musí v Javě implementovat již zmíněné rozhraní `Scriptable`, obsahuje i 5 primitivních datových typů. Tyto datové typy musí být mapovány do Javy na odpovídající třídy. Jakým způsobem je mapování docíleno, je znázorněno v tabulce 3.1.

Typ v JavaScriptu	**Reprezentace v Javě**
`Undefined`	`org.mozilla.JavaScript.Undefined`
`Null`	`null`
`Boolean`	`java.lang.Boolean`
`Number`	`java.lang.Number`
`String`	`java.lang.String`

Tabulka 3.1: Mapování primitivních typů JavaScriptu do Javy [15]

3.1.3 Princip kompilátoru a interpretu

V předchozí kapitole jsme zmínili sadu nejdůležitějších tříd a rozhraní z veřejného API Rhina. V balíku `org.mozilla.JavaScript` se ovšem nachází i jednotlivé části zajišťující samotnou funkci překladače a interpretu JavaScriptu.

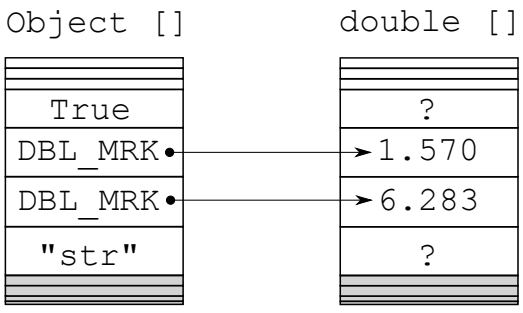
Mezi nejpodstatnější neveřejné třídy v balíku patří např. třída `ScriptRuntime` obsahující metody využívané interpretem a optimalizátorem jazyka pro generování mezikódu Javy. Formát mezikódu lze konkrétně nalézt v balíku `src.org.mozilla.classfile`. Překlad JavaScriptu do mezikódu probíhá běžně za běhu programu. Explicitní překlad skriptu a vytvoření `.class` souboru lze vynutit zavoláním překladače JavaScriptu příkazem:

```
java org.mozilla.JavaScript.tools.jsc.Main script.js
```

Vlastní překladač do mezikódu Javy je velkým specifikem Rhina. Pro každý zkompilovaný skript je vygenerována přesně jedna odpovídající třída v Javě, která implementuje rozhraní `Script`. Skript lze potom vykonat zavoláním metody `exec()` rozhraní `Script`.

Interaktivní (nekompilovaný) režim interpretu JavaScriptu zajišťuje třída `Interpret`, která je umístěná také v hlavním balíku. Interpret JavaScriptu v Rhinu je klasický zásobníkový automat. Pro reprezentaci mezikódu se používá instrukce s proměnlivou délkou. Instrukce se skládá z operačního kódu a případných operandů. Prováděcí smyčku interpretu implementuje metoda `interpretLoop()`, která přijímá aktuální rámec volání a kontext, ve kterém interpretování probíhá. Rámec volání obsahuje typické položky: zásobník hodnot, pole symbolů a aktuální scope objekt.

Zajímavostí interpretu je zavedení dvou druhů zásobníků – hlavního a číselného. Hlavní zásobník je určen pro obecné hodnoty, tzn. pro hodnoty všech typů JavaScriptu. Číselný zásobník slouží pro uložení hodnot JavaScriptového primitivního typu `Number`. Hlavní zásobník je implementován v Javě jako pole objektů `Object` a číselný jako pole primitivního typu `double`. Mezi hlavním a číselným zásobníkem probíhá mapování. Pokud je v hlavním zásobníku uložená unikátní hodnota `DBL_MRK`, tak bude čtení odpovídajících hodnot pokračovat ve druhém zásobníku (obrázek 3.1). Tato technika byla zavedena čistě z důvodu optimalizace, aby nedocházelo ke zbytečnému „zabalení“ primitivního typu do objektu a aby se šetřila alokace paměti pro objekt a případná dereference ukazatelů.



Obrázek 3.1: Znázornění koexistence hlavního a číselného zásobníku [15]

Interpret JavaScriptu přijímá mezikód jazyka, který je generován metodou `compile()` třídy `CodeGenerator`. Generátor mezikódu pracuje s abstraktním syntaktickým stromem tvořeným z prvků definovaných v balíku `org.mozilla.JavaScript.ast` a generovaných třídou `Parser`.

3.2 Nashorn

Nashorn je projekt, který si klade za cíl vyvinout zcela nový, odlehčený a výkonný JavaScriptový runtime pro nativní JVM [13]. Výstupem projektu by měla být schopnost spouštět JavaScriptový kód příkazem `jrunscript`, která podléhá specifikaci JSR 223[6]. Oproti poměrně letitému JavaScriptovému interpretu Rhina vyvinutému pro tehdejší JVM, se Nashorn zaměřuje na nové techniky používání dynamických jazyků v Javě a vychází ze specifikace tzv. Da Vinci stroje (JSR 292[7]) [13].

Jádro interpretu Nashorn vychází ze specifikace ECMAScript verze 5.1. Skripty jazyka JavaScript lze v Javě vytvářet užitím klasického Java API pro skriptování popsaného blíže v kapitole 3.3. Konkrétní instanci JavaScriptového jádra `ScriptEngine` získáme vyhledáním odpovídajícího slova (např. „nashorn“) v továrně `ScriptEngineManager`. Pokud chceme spustit JavaScriptový skript, můžeme toho dosáhnout příkazem `jrunscript`, specifikací skriptovacího jazyka přepínačem `-l` a předáním skriptu přepínačem `-e` nebo souboru se skriptem přepínačem `-f`.

3.3 Skriptovací Java API

V Javě byly postupem času implementovány všechny nejznámější skriptovací jazyky používající různá a vlastní rozhraní. To bylo základním podmětem pro vytvoření obecného a přenositelného skriptovacího Java API, které umožňuje propojit skripty s aplikací Javy. API muselo být napsáno velmi abstraktně, aby bylo aplikovatelné na jakýkoliv rozmanitý skriptovací stroj, ale zároveň muselo zahrnout všechny specifické aspekty známých skriptovacích jazyků [27].

Skriptovací Java API vychází ze specifikace JSR 223. Tato specifikace definuje standardní framework a API pro tvorbu skriptů a jejich vkládání do Java aplikací. Specifikace ovšem nedefinuje jaký jazyk pro skriptování musíme zvolit. Proto můžeme použít pro skriptování jakýkoliv jazyk, který je kompatibilní s JSR 223. Použitím standardního API máme možnost psát skripty kompatibilní s JSR 223.

Implementace Java API v JDK od Oraclu vychází z frameworku BSF [1]. Tento framework byl vyvinut firmou IBM a ve verzi 2.x poskytoval základní množinu tříd umožňujících podporu skriptovacích jazyků v aplikacích Javy. S příchodem skriptovacího Java API byl vydán BSF framework verze 3.x, který se poté stal součástí zmíněného JDK.

Základní Oracle implementace JDK 6 a 7 skriptovacího API obsahuje upravený již zmíněný JavaScriptový stroj vycházející z knihovny Rhina. Zahrnutá Oracle verze knihovny Rhina byla omezena o některé vlastnosti, které by poskytovalo přímé použití knihovny. Z důvodu bezpečnosti byl zakázán kompilátor do mezikódu Javy. V JDK nelze najít ani pomocné nástroje příkazové řádky distribuované Rhinem od Mozilly, jako jsou např.: JavaScriptový shell, debugger apod. Použitá implementace Rhina umožňovala definovat abstraktní třídy a implementovat vícenásobná rozhraní pomocí adaptéru `JavaAdapter`. Oracle nahradil tento adaptér vlastní třídou, která umožňuje implementaci pouze jednoduchých rozhraní.

[6] JSR 223 – specifikace definující framework pro vkládání skriptů do zdrojového kódu Javy
[7] JSR 292 – specifikace pro rozšíření JVM o nativní podporu dynamických jazyků

3.3.1 Architektura skriptování

Skriptovací API je umístěné v balíčku `javax.script`. Skriptování je poměrně přímočaré. Abychom mohli začít psát skripty, vždycky nejprve musíme:

1. vytvořit objekt třídy `ScriptEngineMager` – objekt vyhledává skriptovací enginy specializující třídu `ScriptEngine` v načtených JAR knihovnách. Objekt slouží jako to tzv. továrna na skriptovací enginy;
2. získat instanci skriptovacího enginu `ScriptEngine` – instance skriptovacího enginu je získána z továrny zavoláním např. metody `getEngineByName()`.

Jakmile máme získanou instanci skriptovacího enginu, můžeme již vykonávat, definovat, nebo upravovat skripty.

Nejdůležitější metodou rozhraní `ScriptEngine` je bezpochyby metoda `eval()`, která umožňuje spouštět skripty. Skript může být předán jako řetězec nebo může být umístěn v souboru, streamu, či jiném zdroji. V závislosti na tom, zda daný skriptovací jazyk umožňuje vyhodnocování skriptu s výsledkem, tak metoda `eval()` vrácí i výsledek skriptu.

Myšlenkou standardního skriptovacího Java API je mít co nejvíce společných rysů skriptovacích jazyků specifikovaných v rozhraní `ScriptEngine` a nemít zde žádný rys, který by některý skriptovací jazyk nemohl poskytnout. Proto všechny speciální vlastnosti konkrétních skriptovacích jazyků musí být implementovány v odlišných rozhraních. Tento návrh zajišťuje minimum změn těchto rozhraní v budoucnu.

Mezi nejdůležitější přídavná rozhraní řadíme:

- `Invocable` – rozhraní implementované enginy umožňující volat funkce/metody skriptu. Pro volání funkce používáme `invokeFunction()`, pro metody `invokeMethod()`;
- `Compilable` – rozhraní implementované enginy, které jsou schopny kompilovat skript do svého mezikódu.

3.3.2 Data binding

Důležitým úkolem pro funkci frameworku skriptovacího Java API, je zajistit sdílení dat mezi hostovanou aplikací Javy a skriptovacím strojem. Framework BSF umožňoval sdílení proměnných pouze po náležité registraci v manažeru skriptovacích enginů, k němuž měly přístup všechny skriptovací enginy [27]. Skriptovací Java API model data bindingu vylepšuje tím, že udržuje sdílené proměnné pomocí kontextu, ve kterém je skript spouštěn [27].

Proměnné jsou seskupeny a uloženy ve jmenných prostorech (scope). Abstrakce uchovávání proměnných ve scope je dána rozhraním `javax.script.Bindings`, přičemž se nejedná o nic jiného, než o jednoduché mapování názvu proměnné na náležitý objekt – `Map<String, Object>`. Každý kontext skriptu obsahuje právě množinu takovýchto jmenných prostorů.

Specifikace uvádí dva základní jmenné prostory:

- **Enginový scope** – proměnné, které se vážou k tomuto scope nejsou viditelné jiným strojem;
- **Globání scope** – ke všem proměnným tohoto scope mají přístup všechny enginy vytvořené danou instancí enginového manažeru.

Pokud chceme vytvořit novou proměnnou, dosáhneme toho zavoláním metody `put()` objektu `Bindings`. Proměnnou v enginovém scope lze definovat následovně:

1. Nastavením vlastního objektu `Bindings` – vytvořit mapovací objekt `Bindings`, kde vytvoříme naši proměnnou. Tento mapovací objekt nastavíme v enginu buď metodou `setBindings()`, nebo případně až za běhu skriptu parametrem v metodě `eval()`;
2. Využitím enginového objektu `Bindings` – získáme výchozí mapovací objekt enginu metodou `getBindings()`, do kterého následně vložíme naši proměnnou.

Získání hodnoty proměnné se docílí obdobně – pouze metodou `get()`.

Základní vlastností frameworku je uchovávání stavu proměnných po vykonání skriptu. Všechny proměnné, které jsou ve skriptu inicializovány, se také automaticky vloží do odpovídajícího objektu `Bindings`. Po skončení skriptu jsou vytvořené proměnné dostupné v odpovídajícím scope, kde byly vytvořeny. Tento stav scope zůstává i do dalšího vyhodnocení metodou `eval()`.

Jmenné prostory jsou jedním stavebním prvkem třídy `SimpleScriptContext` – kontextu skriptu. Všechny stavové objekty, které jsou pro běh skriptu podstatné, jsou uložené právě v této třídě. Dokonce i výše popsaný globální a enginový scope je uchováván zde. Jmenné prostory jsou zde uloženy ve formě listu čísel, které identifikují jednotlivé scope. Čísla ve skutečnosti představují prioritu scope. Nízká čísla popisují scope s vysokou prioritou a vysoká čísla s nízkou prioritou. Ve vyhodnocování skriptu to znamená, že pokud např. hledáme proměnnou, která je ve dvou scope zároveň, tak bude navrácena pouze ta proměnná ze scope, který má vyšší prioritu. Enginovému scope je přiřazena priorita 100, globálnímu scope priorita 200. Kromě dvou základních scopů si můžeme definovat i vlastní jmenný prostor. Pro vytvoření vlastního jmenného prostoru ovšem nestačí nastavit tento scope pomocí `setBindings()` ve třídě `SimpleScriptContext`, protože třída přijímá pouze dva základní kontexty. Proto musíme definovat svůj vlastní kontext implementující rozhraní `ScriptContext`.

Kapitola 4

Návrh

Dle zadání této práce máme implementovat podporu pro klientský JavaScript do experimentálního vykreslovacího stroje CSSBox napsaném v jazyku Java. Výsledná aplikace by měla být opět psána v čistém jazyce Java bez použití nativních knihoven, tudíž by neměla narušovat přenositelnost stroje CSSBox.

Z analýzy problematiky jednotlivých existujících strojů JavaScriptu v jazyce Java víme, že v současné době nemáme příliš mnoho na výběr. Jediným použitelným řešením pro skriptování, zejména pro svou podporu hostování objektů Javy ve skriptech JavaScriptu, je pouze knihovna Rhino. Interpret YAJI nebude použit z důvodu jeho malé flexibility a podporovatelnosti. Interpret Nashorn nebyl během návrhu řešení této práce stále oficiálně vydán. Aby se v budoucnu ovšem dalo lehce přejít případně na jiný JavaScriptový stroj, budeme pro implementaci podpory klientského JavaScriptu používat a skriptování z jazyka Javy výhradně standardní skriptovací Java API.

V návrhu se konkrétně zaměříme na problematiku zpracování skriptů z HTML dokumentu (kapitola 4.1) a uvedeme možnost jak zařadit podporu pro skriptování do projektu SwingBox (kapitola 4.2). V kapitole 4.3 naznačíme techniku automatických instalací rozšíření do hlavního scope JavaScriptu. Kapitola 4.4 se bude zabývat spíše experimentální problematikou a to implementací základního zabezpečení do skriptovacího stroje Rhino. V závěru návrhu (kapitola 4.5) budeme demonstrovat integraci rozšíření klientského JavaScriptu do projektu SwingBox.

Návrh byl proveden s ohledem na již existující třídy projektu CSSBox a ostatních knihoven. Návrh vychází z veřejných rozhraní tříd a není-li to zapotřebí, tak se nesnaží již existující třídy znovu implementovat.

Očekávanými závislostmi projektu budou knihovny a projekty:

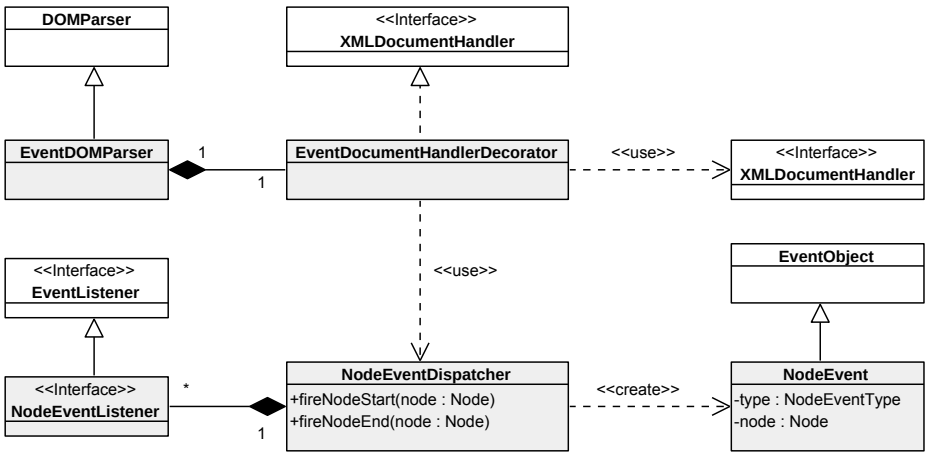
- **CSSBox** [3] – (X)HTML renderovací knihovna,
- **CSSParser** [9] – parser kaskádových stylů,
- **SwingBox** [21] – prohlížeč webových stránek využívající CSSBox,
- **Rhino** [14] – skriptovací stroj JavaScriptu,
- **NekoHTML** [4] – parser HTML dokumentu.

Projekt bude vyvíjen na systému Windows 8 v prostředí Eclipse s JDK 1.7.0_25. Za cíl si projekt klade poskytnout zpětnou podporu pro JRE/JDK 6.

4.1 Provádění skriptů v HTML dokumentu

Klientské skripty jsou v HTML dokumentu specifikovány elementy `<script>`. Jak bylo uvedené v kapitole 2.1, lze skripty vykovávat ihned po dokončení parsování značky se skriptem a nemusí se čekat, dokud je celý dokument načten.

V současné době není v rámci projektu CSSBox podpora pro manipulaci s dokumentem, který je právě čten. Aby bylo možné implementovat podporu pro skriptování přesně podle HTML specifikace, budeme muset implementovat vlastní parser dokumentu. Základní navrženou techniku implementace parseru dokumentu lze spatřit na obrázku 4.1.



Obrázek 4.1: Diagram tříd parseru dokumentu `EventDOMParser`

Podle návrhu bude nutno definovat vlastní parser `EventDOMParser`, který vychází z parseru `DOMParser` knihovny NekoHTML. Tento parser bude průběžně zpracovávat vstupní dokument a informovat o aktuálním stavu parsování pomocí speciálních událostí. Události budou nést informaci, zda bylo zahájeno či ukončeno parsování určitého uzlu. Pokud parsování bylo dokončeno, bude si moci naslouchající objekt vyzvednout rozparsovaný uzel.

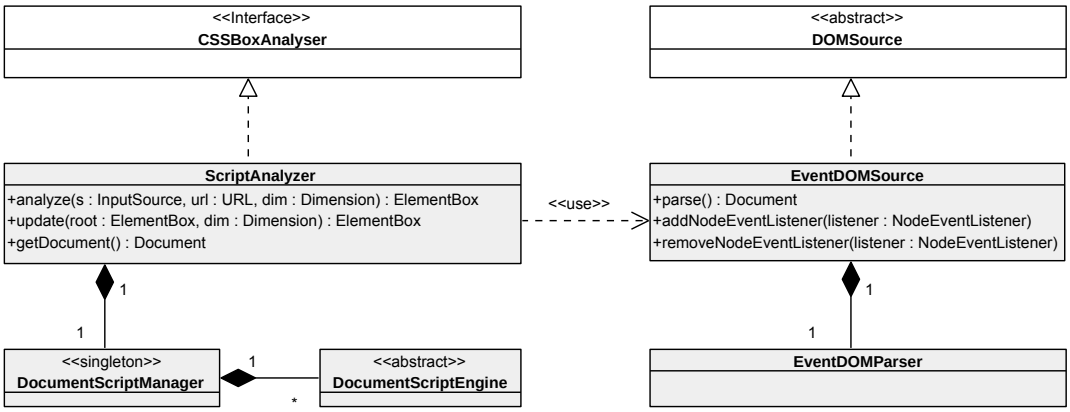
Knihovní třída `DOMParser` využívá pro vytváření dokumentu implementované rozhraní `XMLDocumentHandler`, které obsahuje obslužné metody informující o aktuálním stavu parsování. V rámci těchto obslužných metod dochází k vytváření jednotlivých uzlů dokumentu a konstruování výsledného stromu dokumentu. Současnou implementaci objektu, který zajišťuje konstrukci stromu dokumentu lze získat z konfigurace parseru.

Odchytávání událostí během parsování bude umožněno přenastavením aktuální implementace instance třídy `XMLDocumenthandler` v konfiguraci parseru `DOMParser`. Naším cílem bude v rámci volání obslužných metod zavolat i naši vlastní událost. Abychom ovšem nemuseli předefinovat celou funkčnost konstrukce dokumentu, budeme využívat starý knihovní `XMLDocumentHandler` získaný z konfigurace. Upravený `EventXMLDocumentHandler` bude knihovní handler pouze dekorovat a zajišťovat propagaci některých událostí výše – do třídy `EventDOMParser`.

Aby byl návrh kompletní, byla navržena i třída `NodeEventDispatcher` vykonávající události `NodeEvent`. Všichni zájemci o naslouchání událostí si budou muset zaregistrovat naslouchající objekt ve třídě `EventDOMParser`, který dále tento objekt zaregistruje i ve třídě `NodeEventDispatcher`. Objekty odchytávající události parseru budou implementovat rozhraní `NodeEventListener` vycházející ze standardního rozhraní Javy `EventListener`.

4.2 Podpora skriptování v projektu SwingBox

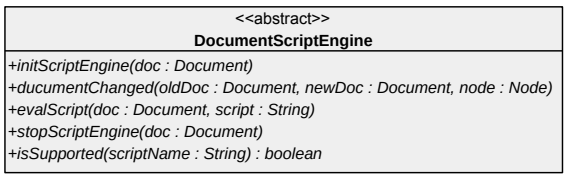
Jak bylo popsáno v kapitole 2.3, projekt SwingBox umožňuje zobrazovat a procházet webové stránky. Pro přidání podpory skriptování do tohoto projektu (viz kapitola 4.5) bude nutno definovat vlastní třídu `CSSBoxAnalyzer`. SwingBox používá tuto třídu pro parsování vstupního dokumentu a převod dokumentu na vykreslovaný `ElementBox`.



Obrázek 4.2: Diagram tříd znázorňující podporu JavaScriptu v projektu SwingBox

Projekt `CSSBox` definuje pro zdroj dokumentu třídu `DocumentSource`, pro čtení dokumentu ze zdroje dokumentu třídu `DOMSource`. Abychom jsme se drželi zavedené praktiky pro čtení dokumentu s událostmi, v návrhu definujeme třídu `EventDOMSource`, která pouze vnitřně používá v kapitole 4.1 popsaný `EventDOMParser`.

Největší roli při invokaci skriptu hraje v návrhu `ScriptAnalyzer`, který má u událostního parseru zaregistrovaný vlastní `EventNodeListener`. Při příchodu události o zpracování elementu `<script>`, přečte `ScriptAnalyzer` skript v jeho těle, případně načte skript z URL adresy uvedené v `src` atributu. Pokud skript neobsahoval atribut `defer`, tak se dále nechá ihned vykonávat ve stroji `DocumentScriptEngine`. Stroj, který má být použit pro aktuální skript, je získán z instance třídy `DocumentScriptManager` na základě atributu `type` elementu `<script>`, nebo z výchozího uvedeného stroje v `<meta>` značce (viz kapitola 2.1).

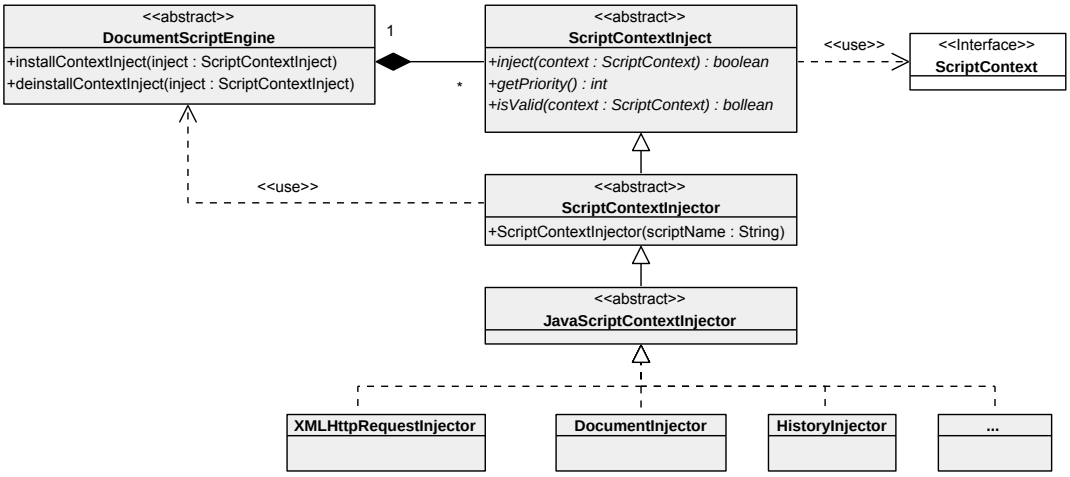


Obrázek 4.3: Diagram abstraktní třídy `DocumentScriptEngine`

Pokud je skript vykováván poprvé v rámci uvedeného dokumentu, tak dojde u třídy `DocumentScriptEngine` (obrázek 4.3) i k zavolání metody `initScriptEngine()`, která vytvoří globální scope pro vykonávání skriptů. V globálním scope jsou poté vykonávány všechny skripty na stránce. Skriptovací stroj uchovává jednotlivé zaregistrované dokumenty a jejich vykonávací globální scope, dokud není webová stránka přenačtena a dokument není změněn. O zničení dokumentu bude stroj informován metodou `stopScriptEngine()`.

4.3 Injektování objektů do hlavního scope

Během implementace podpory klientského JavaScriptu do projektu CSSBox se předpokládá vytvoření velkého množství tříd, které budou implementovat jednotlivé aspekty JavaScriptu pro prohlížeč. Předpokládá se, že nové třídy budou vyvíjeny postupně a iterativně přidávat jednotlivé funkčnosti. Bylo by výhodné oddělit jednotlivé nové funkce a odstranit potřebu přímého zásahu do třídy `DocumentScriptEngine` s příchodem nového rozšíření.



Obrázek 4.4: Diagram tříd znázorňující podporu injektování rozšíření JavaScriptu

K separaci nových implementovaných vlastností JavaScriptu byla navržena technika injekce funkčnosti do vytvořeného kontextu (obrázek 4.4). Pro přidání nové funkčnosti do globálního scope kontextu bude nutno realizovat abstraktní třídu `ScriptContextInject`. Samotnou injekci do globálního scope bude provádět metoda `inject()`. Aby skriptovací stroj věděl, jaká rozšíření má do svého scope injektovat, měla by být jednotlivá rozšíření registrována u skriptovacího stroje metodou `installContextInject()`. Injekce bude prováděna vždy po vyhodnocení některého skriptu nebo po inicializování nového dokumentu. Pořadí jednotlivých injekcí bude v základní implementaci určovat metoda `getPriority()`. Zda je injekce validní (proveditelná), bude získáváno metodou `isValid()`. Příkladem, kdy injekce by byla neproveditelná, by mohla být např. situace, kdy není ještě dokument úplně načten a některé rozšíření by záviselo právě na existenci dokumentu.

Neustálá instalace nové injekce rozšíření do skriptovacího stroje s příchodem nového rozšíření by se postupem času nejevila jako dobré řešení. Bylo by výhodné, aby s novým rozšířením došlo i k automatické instalaci tohoto rozšíření do skriptovacího stroje, tzn., abychom dosáhli obrácení řízení – IoC (Inversion of Control). Pro tento účel byla navržena třída `ScriptContextInjector`, která kromě metod samotné injekce bude provádět i registraci rozšíření v odpovídajícím skriptovacím stroji. Bude-li to implementačně vhodné, bude se k získání reference registru skriptovacích strojů `DocumentScriptManager` z objektů `ScriptContextInjector` používat injektáž závislosti – DI (Dependency Injection). Pro podporu DI je v plánu použít knihovnu Google Juice [6]. Ve třídě `ScriptContextInjector` se v závislosti na typu skriptu vyhledá v registru skriptovacích strojů odpovídající stroj, ve kterém proběhne registrace samotné injekce rozšíření.

Pokud celý návrh injektování shrneme, tak automatická registrace injekce rozšíření bude

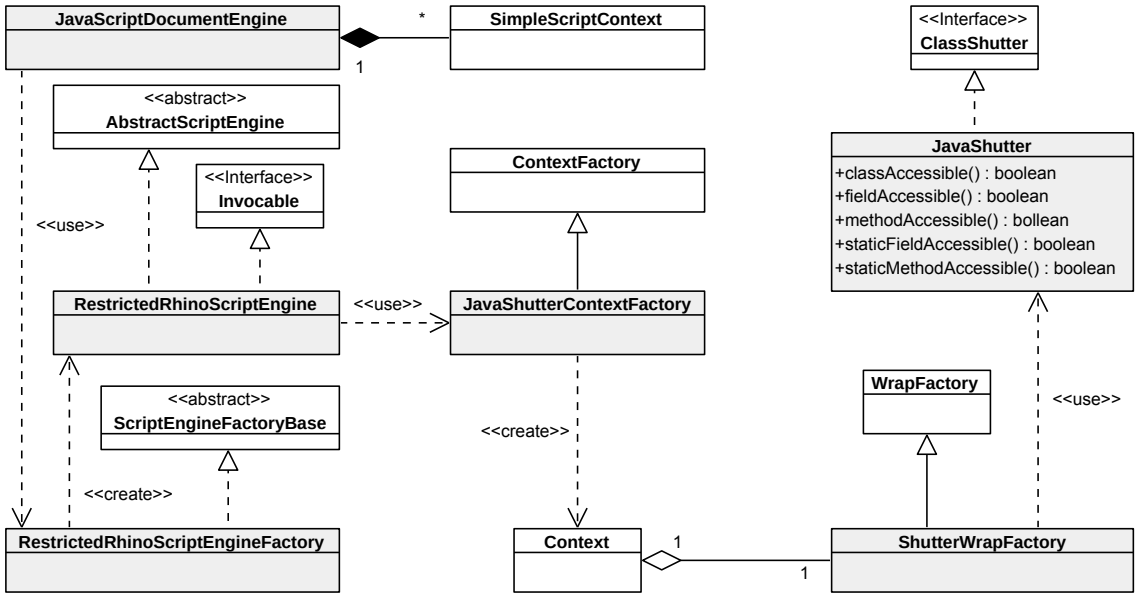
probíhat v abstraktní třídě `ScriptContextInjector`. O tom, do jakého skriptovacího stroje by se měla injekce rozšíření vložit, bude informovat pro jazyk JavaScript např. třída injektoru `JavaScriptContextInjector`. Samotná injekce rozšíření bude přitom implementována až specifickými injektory, jako jsou např. `XMLHttpRequestInjector`, `ConsoleInjector`, apod.

4.4 Implementace stroje JavaScriptu podle normy JSR 223

Poslední iterací v integraci JavaScriptového stroje do projektu CSSBox by bylo přizpůsobit skriptovací stroj tak, aby splňoval bezpečnostní předpoklady pro běh v prohlížeči. Budeme muset zabránit zejména přístupu do tříd a balíků Javy, které nenesou žádnou funkční hodnotu pro implementaci klientského JavaScriptu. Z důvodu snazšího přechodu na jiný skriptovací stroj, např. Nashorn, by bylo výhodné, aby byl stroj implementován podle standardu JSR 223. Nashorn je vyvíjen se standardním rozhraním již od základu.

Pro tvorbu takového stroje by se nabízelo využít již existující stroj implementovaný v Java API, který se nachází v balíku `sun.org.mozilla.JavaScript`. Třídy Java API jsou ovšem finální nebo skryty, což brání v jejich znovupoužití. Nezbývá nám než implementovat vlastní skriptovací stroj, který rozšiřuje základní třídu pro skriptovací stroje `AbstractScriptEngine`. Tento stroj bude pro implementaci samotného skriptování využívat externí knihovnu Rhino. Rozhodnutí pro externí knihovnu bylo učiněno z důvodu, abychom nebyli závislí na dodávané knihovně v JDK v balíku `sun.org.mozilla.JavaScript.internal`. Osamostatněním od implementace JDK získáme kompletní správu nad verzí knihovny a nebudeme muset řešit případné problémy s příchodem JDK 8, kde již Rhino zřejmě nebude ani v rámci základní distribuce JDK.

Způsob, kterým by mohl být zhotoven JavaScriptový stroj s implementovanými bezpečnostními funkcemi je navržen na obrázku 4.5:



Obrázek 4.5: Diagram tříd zabezpečeného skriptavacího stroje JavaScriptu

Skriptovací stroj spravující jednotlivé načtené webové dokumenty reprezentovaný třídou `JavaScriptDocumentEngine` bude používat instanci námi upraveného skriptovacího stroje nazvaného `RestrictedRhinoScriptEngine`. Aby byla splněna specifikace JSR 223, tak bude tento stroj vytvářen pomocí továrny, která bude realizovat abstraktní třídu Java API `ScriptEngineFactoryBase`. Třída `RestrictedRhinoScriptEngine` bude implementovat abstraktní třídu pro všechny skriptovací stroje `AbstractScriptEngine`. Jelikož JavaScript umožňuje i volat funkce, bude dodatečně implementováno i rozhraní `Invocable`.

Námi implementovaný stroj bude staticky vytvářet objekty nejvyšší úrovně a standardní objekty JavaScriptu, které budou společné pro globální scope a pro každé vykonávání skriptu. Staticky bude vytvořena i továrna `JavaShutterContextFactory` pro vytváření kontextu, která implementuje `ContextFactory`. Továrna je automaticky volána knihovnou Rhino pokaždé, když vstoupíme do nového kontextu metodou `Context.enter()`. Továrna kontextu bude instalovat do kontextu vlastní obalující továrnu `ShutterWrapFactory` a objekt třídy `JavaShutter` sloužící pro omezení přístupu do jazyka Javy. Objekt omezující bezpečnost bude viditelný i z továrny pro obalování objektů Javy.

Důvodem pro předefinování továrny obalující objekty, která je volána pro každou třídu, objekt, metodu, primitivní typ Javy použitý v kódu JavaScriptu, je právě přidání podpory rozmělnění bezpečnosti. Samotným účelem obalující továrny je dodání rozhraní `Scriptable` do objektů Javy, tak aby byly viditelné z kódu JavaScriptu. Ve vlastní implementaci továrny budeme provádět před obalením položky ověření, zda je položka viditelná. Pokud položka nemá být viditelná, tak nedojde k požadovanému obalení položky, takže položka z kódu JavaScriptu nebude viditelná.

Knihovna Rhino již obsahuje rozhraní `ClassShutter` pro zajištění bezpečnosti přístupu do Javy. Rozhraní `ClassShutter` však obsahuje pouze jednu metodu, která ověří, zda je požadovaná třída viditelná. Navrhnutá architektura na obrázku 4.5 ovšem uvádí rozmělnění zabezpečení do menších položek, jako jsou povolené metody, atributy aj. Takovéto rozmělnění bude zajištěno objektem `JavaShutter`, který bude poskytovat informaci o tom, zda je přistupovaná třída, metoda, vlastnost apod. viditelná z kódu JavaScriptu. Registrace viditelných položek bude implementačně prováděna manuálně zavoláním určité metody nad objektem `JavaShutter`. Budeme mít ale i možnost nechat viditelnost odvodit automaticky podle anotace položky. Pro viditelné položky Javy zavedeme vlastní anotaci `@JSVisible`.

Druhým bezpečnostním omezením bude omezení adaptéru zabezpečujícího implementaci rozhraní a abstraktních tříd Javy – `JavaAdapter`. Předpokládá se, že tato funkčnost bude kompletně zakázána.

Po vyřešení bezpečnosti budeme muset implementovat samotné propojení rozhraní Rhina a specifikace JSR 223. Důležité bude vyřešit data binding pro kontext `ScriptContext` definovaný v této specifikaci. Abychom měli přehled o všech proměnných, které se ve hlavním scope nachází, budeme muset vytvořit třídu vlastního scope. Scope (viz kapitola 3.1.2) je klasický JavaScriptový objekt, ve kterém probíhá spouštění skriptu, tudíž implementuje rozhraní `Scriptable`. Rozhraní specifikuje metody pro zápis `put()` a získání atributů `get()` z objektu. Tyto metody jsou během vykonávání skriptu náležitě volány. Zpětného volání metod využijeme k navázání atributů do objektu `Bindings`.

Výše popsané praktiky jsou dostačující pro vytvoření jednoduchého stroje, který vychází ze specifikace JSR 223. Po zavolání vykonání skriptu metodou `eval()`, bude pracovat stroj tak, jak je popsáno v následujících krocích:

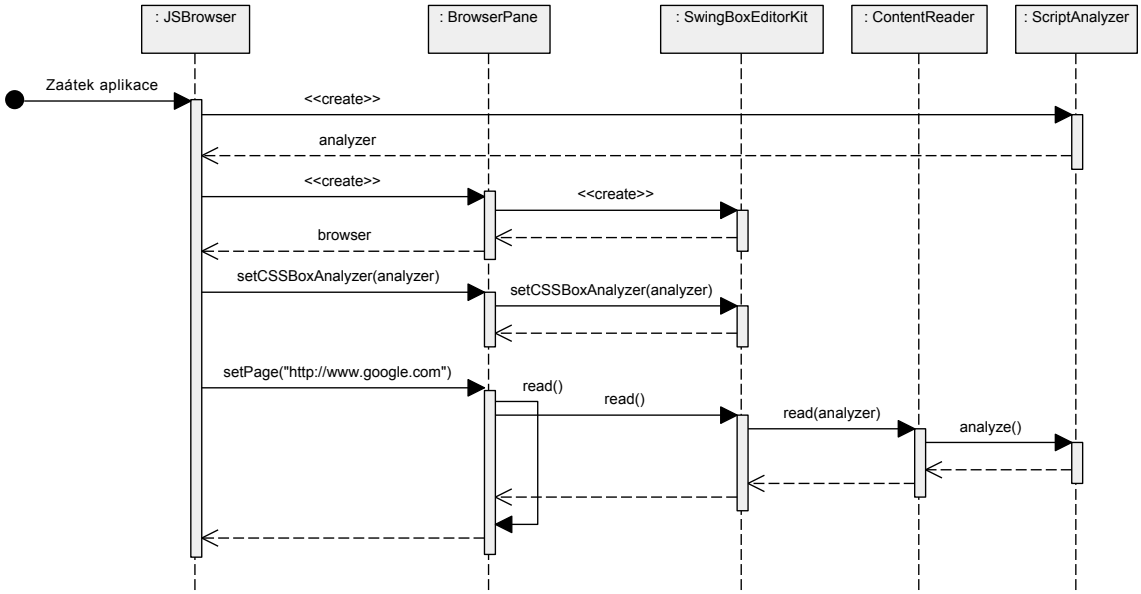
1. Vytvoří se nový kontext zavoláním `Context.enter()`, v rámci něhož dojde i k zavolání metody `makeContext()` třídy `JavaShutterContextFactory`;

2. Vytvoří se nový hlavní scope, který bude nastavovat a číst `Bindings` objekt;
3. Do hlavního scope se vloží standardní a globální objekty, které budou společné pro všechny skripty. Vložení proběhne nastavením prototypu hlavního scope;
4. Spustí se skript nad vytvořeným kontextem a hlavním scopem.

4.5 Integrace podpory JavaScriptu do SwingBoxu

Všechny potřebné náležitosti pro zařazení podpory JavaScriptu do projektu CSSBox byly již navrženy. Návrh proběhl konkrétně pro podprojekt SwingBox, který rozšiřuje výchozí CSSBox o reakci prvků stránky na podněty příchozí od uživatele, tzn. zejména o možnost procházení webového obsahu.

Aby bylo možné zakomponovat JavaScript do projektu SwingBox, bude nutno provést registraci vlastní `CSSBoxAnalyzer` třídy navržené v kapitole 4.2. Kompletní konfigurace, kterou bude nutno provést pro přidání podpory JavaScriptu, je znázorněna na obrázku 4.6.



Obrázek 4.6: Sekvenční diagram znázorňující kroky integrace JavaScriptového stroje

Z uvedeného diagramu je patrné, že pouhá jedna registrace vlastního analyzátoru dokumentu `ScriptAnalyzer` bude pro přidání podpory JavaScriptu dostačující. Projekt SwingBox zejména implementuje vlastní zobrazovací plochu pro webové stránky `BrowserPane`. Zobrazovací plochu využijeme ve finální části projektu pro tvorbu jednoduchého webového prohlížeče a znázornění funkčnosti skriptování v HTML dokumentech. Třída pro zobrazování obsahu HTML stránky využívá pro veškeré čtení nového dokumentu webové stránky třídu `SwingBoxEditrKit`, která následně volá `ContentReader`, jenž je přímo spjat s aktuálně zaregistrovaným `CSSBoxAnalyzer`. Definicí vlastní třídy `ScriptAnalyzer` získáme plnou kontrolu nad zpracovávaným dokumentem a možnost provádění skriptů v HTML dokumentu.

Kapitola 5

Implementace

Následující kapitola pojednává o realizaci rozšíření projektu CSSBox o možnost provádění skriptů v (X)HTML dokumentech navrženém v kapitole 4. Výstupem realizace je volitelné rozšíření knihovny CSSBox v podobě knihovny. Návod na kompilaci knihovny lze nalézt v příloze C. Popis implementace, kterým se tato kapitola bude zabývat, je zaměřen na uvedení nejzákladnějších tříd, metod či funkcí, které implementují nejdůležitější části výsledné knihovny. Podrobný implementační popis lze nalézt v programových dokumentacích na přiloženém DVD (příloha A).

Cílem implementace bylo přidat podporu pro skriptování v (X)HTML dokumentech a zahrnout podmnožinu aktuální specifikace klientského JavaScriptu. Aby bylo možné implementovat klientské skriptovací enginy, bylo zapotřebí vytvořit jednoduché jádro prohlížeče implementující základní aspekty procházení stránek (kapitola 5.1). Pro snadné přidávání nových klientských skriptovacích enginů byla vytvořena abstraktní architektura pro skriptování (kapitola 5.2), ze které vychází i JavaScriptový klientský engine (kapitola 5.3).

Implementace této práce se nachází v podbalících hlavního balíku projektu ScriptBox pojmenovaném `org.fit.cssbox.scriptbox`. Názvy tříd, které tyto podbalíky obsahují, nebudou v této kapitole z důvodu zjednodušení a přehlednosti plně kvalifikovány. Stručné popisy jednotlivých balíků jsou v příloze G.

5.1 Jádro prohlížeče a jeho rozhraní

Jednotlivé implementace klientských skriptovacích enginů využívají různá rozhraní pro komunikaci s prohlížečem a dokumentem, ve kterém jsou jejich skripty vloženy. Těmito rozhraními myslíme např. rozhraní pro objekty `Window`, `Location`, `History`, `Document` aj. Aby bylo možné poskytnout implementaci pro tyto zmíněné rozhraní, bylo nutné implementovat jednoduché jádro pro procházení HTML stránek.

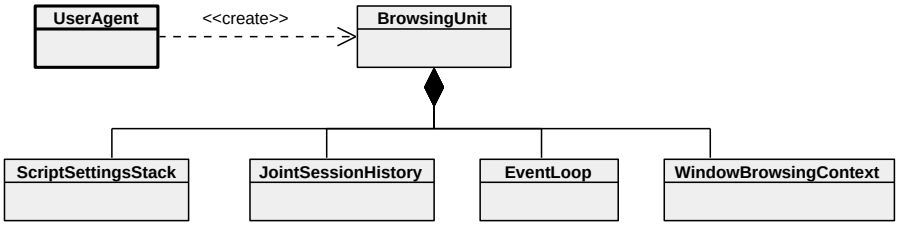
V této kapitole se budeme zabývat obecným jádrem pro procházení HTML stránek. Jelikož současná implementace probíhala primárně pro JavaScriptový engine, popíšeme také korespondující rozhraní, která jsou viditelná v klientském JavaScriptu. Jednotlivá implementovaná rozhraní se snaží v maximální možné míře využívat fundamentální koncept jádra prohlížeče. Samotné jádro je s těmito rozhraními provázáno jen minimálně.

V kapitole 5.1.1 nejprve definujeme základní pohled na prohlížeč jako na prohlížecí jednotky kontextu. Kapitola 5.1.2 pojednává o událostní smyčce, která zapouzdřuje všechny úlohy prohlížeče. Navigace dokumentu a jeho navracení procházením historie je popsáno v kapitolách 5.1.5 a 5.1.6. Následující kapitoly dokončují výklad a přidávají další rozhraní, která jsou viditelná z klientského JavaScriptu.

5.1.1 Základní koncepty prohlížeče

Prohlížečem budeme od této chvíle rozumět uživatelského agenta, angl. user agent, ať již bude, nebo nebude sloužit k prohlížení webu. Tento pojem bude reprezentovat instanci celého funkčního jádra, které umožňuje navigování HTML dokumentů a jiných podporovaných zdrojů.

V implementaci představuje uživatelského agenta třída `UserAgent`, která shromažďuje nastavení a umožňuje jednotný přístup ke sdíleným datům. Hlavní funkcí uživatelského agenta je otevírání a zavírání procházecích jednotek `BrowsingUnit` (obrázek 5.1), neboli jednotek, které zapouzdřují kontext na nejvyšší úrovni (okno, prohlížecí panel) pro zobrazování dokumentu. Hlavní procházecí kontext repezentuje třída `WindowBrowsingContext`.



Obrázek 5.1: Diagram tříd procházecí jednotky vytvářené uživatelským agentem

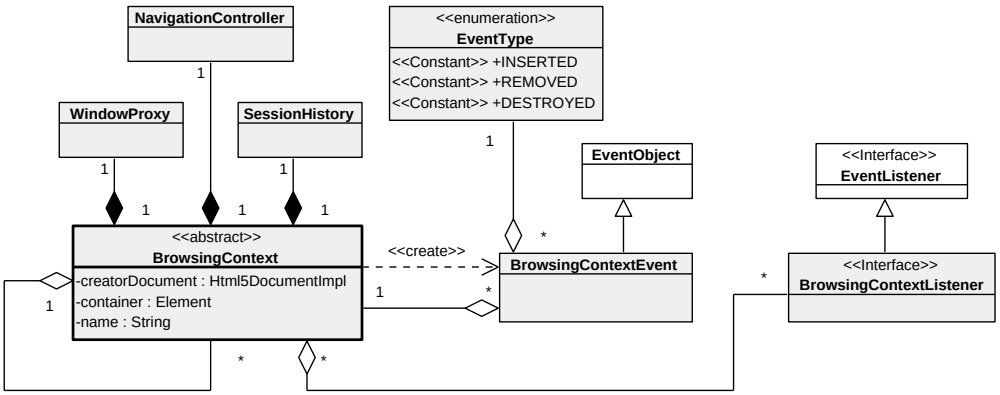
Uvnitř jednotek je dále uchovávána reference na událostní smyčku (kapitola 5.1.2), historii procházení (kapitola 5.1.6) a zásobník právě prováděných skriptů (kapitola 5.2.1), které jsou společné pro všechny obsažené procházecí kontexty. Všechny výše zmíněné objekty jsou vytvářeny v rámci konstrukce procházecí jednotky.

Výše jsme zmínili, že procházecí jednotka vytváří procházecí kontexty. Tímto kontextem rozumíme prostředí, ve kterém jsou prezentovány dokumenty uživateli. Během navigování stránek dochází ke změně dokumentů, nikoliv však procházecího kontextu, který je po celou životnost (např. okna prohlížeče) neměnný. Procházecí kontexty poskytují jakýsi obal nad navigovanými dokumenty a informují o právě aktivním dokumentu.

Procházecí kontexty mohou být různě hierarchicky stromově zanořeny. Kořen stromu je umístěn vždy v procházecí jednotce, pokud zanedbáme pomocné procházecí kontexty `AuxiliaryBrowsingContext`. Zanoření vzniká zejména díky procházecím kontextům vytvořených přes `<iframe>` značky, které mohou obsahovat zanořené dokumenty.

Pokud je uživatelským agentem agent, který obsahuje uživatelské prostředí, tak má procházecí kontext velmi blízké provázání právě na toto uživatelské prostředí, ve kterém je zobrazován dokument. Proto byl kontext zvolen i pro definici rozhraní umožňujícího základní přístup k uživatelskému rozhraní z klientských skriptů. V kontextu se nachází rozhraní například pro navigační pole, statusový panel, posuvníky aj. Podle HTML 5 jsou tato uživatelská rozhraní viditelná skrze `BarProp` rozhraní. Abychom odstínili vytváření nových „schopnějších“ procházecích kontextů při každé potřebě na změnu uživatelského rozhraní, je možné tato rozhraní nastavovat v obecných a již implementovaných procházecích kontextech pomocí odpovídajících setterů. Jelikož základní uživatelský agent `UserAgent` žádné uživatelské prostředí nedefinuje, základní implementace procházecích kontextů neposkytuje ani kompletní implementaci `BarProp` rozhraní. Implementace je provedena pouze formou nekompletních metod, známých angl. pod pojmem stubs, který budeme od této chvíle v textu dále používat.

Abstraktní třídu pro všechny procházecí kontexty představuje třída `BrowsingContext` (obrázek 5.2), která obsahuje metody pro vytváření a hierarchickou strukturalizaci procházecích kontextů a ostatní funkcionalitu společnou pro všechny specifické kontexty. Třída vlastní referenci na historii sezení `SessionHistory` (kapitola 5.1.6), navigační kontroler `NavigationController` (kapitola 5.1.5) a objekt `WindowProxy` (kapitola 5.1.8).

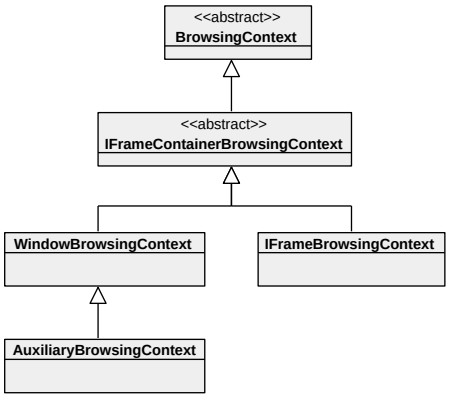


Obrázek 5.2: Diagram tříd procházecího kontextu

Nejdůležitějšími atributy třídy `BrowsingContext` jsou:

- `creatorDocument` – reference na dokument, který vytvořil procházecí kontext;
- `container` – element, v rámci něhož byl vytvořen procházecí kontext;
- `name` – jméno procházecího kontextu.

Všechny specifické implementované procházecí kontexty jsou znázorněny na obrázku 5.3.



Obrázek 5.3: Diagram tříd hierarchie procházecích kontextů

Z třídy kontextu dědí další abstraktní třída `IFrameContainerBrowsingContext`, která je určena pro všechny procházecí kontexty, jež mohou vytvářet zanořené procházecí přes značky `<iframe>`. Další třídy `WindowBrowsingContext` a `IFrameBrowsingContext` reprezentují již konkrétní procházecí kontexty. Třída `WindowBrowsingContext` představuje kontext na nejvyšší úrovni – tedy na úrovni okna. Druhá třída `IFrameBrowsingContext` pak

plní funkci vnořeného kontextu vytvářeného uvnitř `IFrameContainerBrowsingContext` přes značku `<iframe>`. Pomocný kontext `AuxiliaryBrowsingContext` je dalším kontextem na nejvyšší úrovni s tím rozdílem, že není vytvářen procházecí jednotkou jako u kontextu okna. Kontext je naopak vytvářen libovolným specifickým kontextem, který potřebuje otevřít nové okno, vznikajícím např. navigací cíle `_blank`.

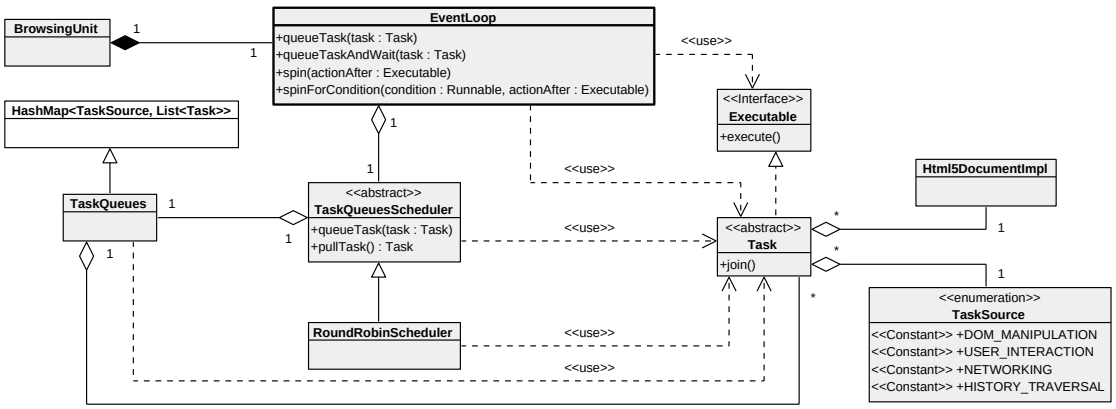
V průběhu implementace se naskytla potřeba pro získávání notifikací z procházecích kontextů při výskytu určité události, a tak byl implementován jednoduchý observer mechanismus nad procházecími kontexty. Procházecí kontext `BrowsingContext` může vyvolat události:

- `INSERTED` – po vložení nového zanořeného kontextu do daného kontextu;
- `REMOVED` – při smazání vnořeného kontext z daného kontextu;
- `DESTROYED` – pokud je daný kontext zrušen.

5.1.2 Událostní smyčka

Abychom mohli pokračovat ve výkladu, je zapotřebí se seznámit s událostní smyčkou, která obsluhuje úlohy uživatelského agenta. Událostní smyčka umožňuje úlohy dle potřeby mazat, různě filtrovat nebo je oddělit podle jejich typu – zdroje úlohy. Smyčka dále sjednocuje přístup k API uživatelského agenta v jednom vlákně, aby nedocházelo k race condition a ulehčilo se řešení výlučných sekcí. Asynchronní žádosti na API, jako je např. navigování stránek nebo procházení historie, v určitém čase vždy zajistí vložení nové úlohy do událostní smyčky.

V implementaci je událostní smyčka reprezentována třídou `EventLoop` a je vytvářena procházecí jednotkou v rámci její konstrukce. Zjednodušený diagram tříd znázorňující funkci událostní smyčky lze vidět na následujícím obrázku 5.4.



Obrázek 5.4: Diagram tříd událostní smyčky

Nejdůležitější metoda smyčky je `queueTask()`, která vkládá nové úlohy `Task` do front úloh `TaskQueues` v závislosti na tom, o jakou frontu se jedná. Rozlišení front je docíleno zdrojem úlohy `TaskSource`. Vkládání úloh a její vybírání zprostředkovává plánovač `TaskQueuesScheduler`.

Kromě operace na vkládání nových úloh obsahuje událostní smyčka různé variace metod na filtraci a mazání úloh. Tyto metody obsahují ve svých názvech prefixy `filter...()`

a `remove...()`. Při konstrukci smyčky se vytváří nové vlákno, které čeká, dokud není naplánována nová úloha plánovačem úloh. Obecně platí, že událostní smyčka je vláknově bezpečná. Pokud si přejeme pozdržet vlákno, které vložilo úlohu do front úloh, lze toho docílit metodou `queueTaskAndWait()`, která interně volá metodu `join()` nad danou úlohou. Pokud bychom použili metodu `queueTaskAndWait()` přímo z vykonávacího vlákna smyčky, došlo by k jeho zablokování. Proto je zapotřebí využít jiných technik, např. zavolání metody `spin()` poté, co je vložena nová úloha.

Metoda `spin()` a její variace s prefixem `spin...()` zajišťují rotaci událostní smyčky. Rotací rozumíme přerušení aktuálně prováděné úlohy a její opětovné vložení, přičemž vložení může být různě zpožděno. Základní metoda na rotaci `spin()` vkládá úlohu ihned do front úloh, zatímco metoda `spinForCondition()` čeká na splnění určité podmínky. Podmínky jsou implementovány rozhraním `Runnable`, jehož metoda `run()` je při žádosti na odložené vložení zavolána ve speciálním vlákně. Po skončení vlákna dojde teprve ke vložení úlohy do front úloh. Přerušení právě probíhající úlohy je zajištěno vyhozením kontrolované výjimky `TaskAbortedException`, která pokud není nikde odchycena, je propagována až do událostní smyčky. Smyčka po zachycení dané výjimky začne považovat právě vykonávající úlohu jako za dokončenou. Úloha, která je vložena rotací smyčky do front úloh, je kopií staré úlohy ovšem s novým vykonávacím tělem, jež je předáváno jako `actionAfter` parametr do metod pro rotace. Rotací událostní smyčky lze využít např. během posouvání na fragment stránky, přičemž stránka není ještě stále kompletní. V tomto případě lze rotací zapříčinit čekání na daný fragment a poté, co je vložen do dokumentu, vložit úlohu, která zajistí posunutí na daný fragment.

Jak víme, každá úloha musí rozšiřovat abstraktní třídu `Task`. Úloha musí být dále také asociována s dokumentem a musí nést zdroj, podle kterého se rozhoduje, do jaké fronty se má úloha zařadit. Položka dokumentu v úloze velmi často slouží k určení cíle úlohy, čehož se využívá při odstraňování úloh pro dokumenty, které vystoupily z procházecího kontextu, např. pokud došlo k navigaci na jiný dokument.

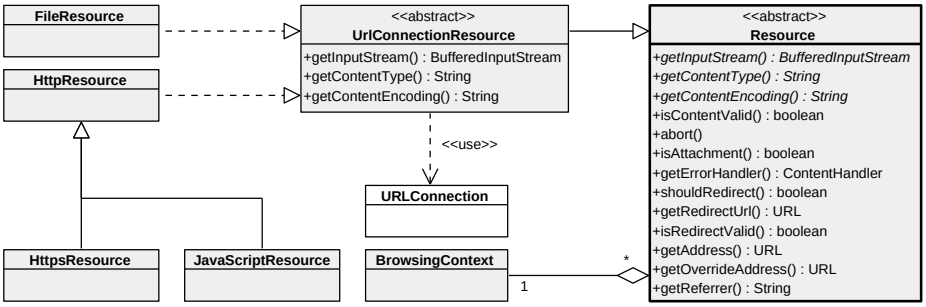
Fronty úloh `TaskQueues` jsou rozšířením hašovací tabulky, která mapuje zdroje úloh na fronty přiřazené k tomuto zdroji. Třída `TaskQueues` přidává nové metody pro intuitivnější práci s frontami úloh – metody pro vyhledávání úloh ve frontách, jejich filtraci a odstraňování.

Cílem plánovače úloh `TaskQueuesScheduler` bylo zajistit oddělené plánování úloh od událostní smyčky a umožnit v budoucnu různé implementace plánování. Tak jako událostní smyčka, má plánovač vlastní vlákno pro plánování. Přes své rozhraní přijímá nové úlohy, které vkládá do front úloh. Ve vlastním vykonávacím vlákně tyto úlohy podle plánovacího algoritmu vybírá a vkládá je do fronty naplánovaných úloh. V současnosti jediná implementace plánovače úloh `RoundRobinScheduler` zajišťuje rotační odebírání úloh z front úloh, přičemž pokud je některý zdroj úlohy neaktivní a jeho fronta je prázdná, je po určitém časovém kvantu vnitřním časovačem odstraněn z pole pro rotaci.

5.1.3 Stahování zdrojů

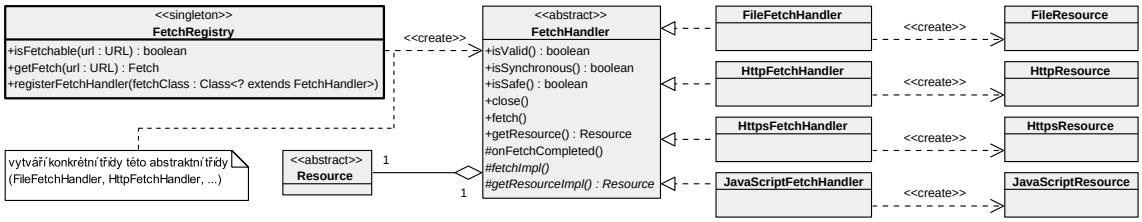
Předtím než přejdeme k popisu dokumentu a jeho parsování, popíšeme v následující kapitole, jak je řešeno jeho stažení. Stahování bylo řešeno obecně pro různé typy komunikačních protokolů a pro různé typy formátů obsahu. Jelikož lze stahovat různé druhy obsahu, nikoliv pouze dokumenty, budeme se v této kapitole bavit pouze o zdrojích. Pokud není obsahem zdroje přímo dokument, jsou obsahy často zpracovávány a do dokumentu dodatečně vkládány. Zpracovávání obsahu zdrojů popíšeme blíže v následující kapitole 5.1.4.

Základní třídou pro všechny zdroje je v implementaci třída `Resource` (obrázek 5.5). Třída poskytuje vstupní datový tok obsahu zdroje, nese základní informace o tom, z jaké adresy byl zdroj získán a typu jeho obsahu. Informuje, zda zdroj je dostupný nebo jestli by mělo být provedeno přesměrování. Pokud je zdroj získáván pomocí `URLConnection` rozhraní, existuje v implementaci bližší reprezentace pro tyto zdroje – abstraktní třída `URLConnectionResource`. Byly implementovány celkem čtyři druhy konkrétních zdrojů: `HttpResource`, `HttpsResource`, `FileResource` a `JavaScriptResource`. V současné době lze tedy stahovat zdroje z URL adres, jež nesou protokoly `file`, `http`, `https` a `javascript`.



Obrázek 5.5: Diagram tříd abstraktní třídy pro všechny zdroje

Pokud hovoříme o stažení zdroje, máme na mysli navázání připojení a podle daného protokolu komunikaci se serverem. Komunikaci se serverem a vytváření konkrétního zdroje zajišťují třídy ovladačů – třídy rozšiřující abstraktní třídu pro ovladače `FetchHandler`. Každý konkrétní ovladač vytváří jeden z výše zmíněných specifických zdrojů (obrázek 5.6). Pro stažení zdroje je vyčleněna metoda `fetch()`, která vnitřně zavolá chráněnou metodu `fetchImpl()`. Pokud byl ovladač vytvořen v synchronním módu, pak tato operace způsobí blokování do doby stažení zdroje. Pro získání nově staženého zdroje slouží metoda `getResource()`, která také volá chráněnou metodu, tentokrát `getResourceImpl()`.

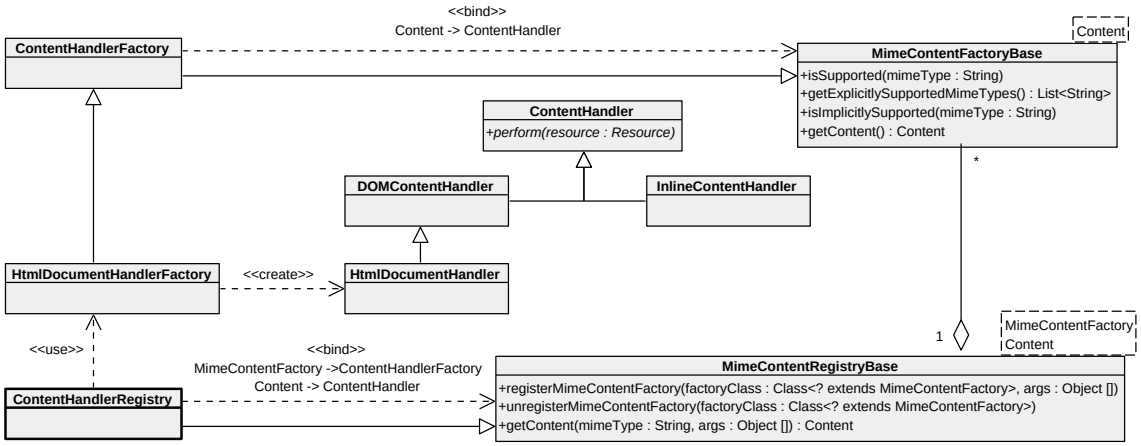


Obrázek 5.6: Diagram tříd registru ovladačů

V praxi při získávání zdroje máme k dispozici pouze jeho URL. Nevíme předem, jaký ovladač pro jeho stažení použít. Přesto bychom chtěli mít jednotné rozhraní, které by převzalo předanou URL a vyhodnotilo, zda je zdroj stažitelný, popř. ho i stáhlo. Za tímto účelem vznikl registr všech ovladačů `FetchRegistry`, který shromažďuje všechny podporované ovladače a umožňuje nad nimi vyhledávat podle protokolu předané URL. Registr registruje třídy ovladačů a vytváří na ně asociace v závislosti na tom, jaké protokoly podporují. Podporované protokoly jsou získávány ze speciální anotace třídy `FetchHandlerPreamble`, která musí být přítomna u každé třídy registrovaného ovladače.

5.1.4 Zpracovávání obsahu zdrojů

V předešlé kapitole 5.1.3 bylo popsáno, jak lze získávat zdroje. Zdroje musí být ovšem v závislosti na typu obsahu dále zpracovávány, převedeny na dokument a až poté mohou být zobrazeny. Pro zpracovávání byla zvolena podobná návrhová myšlenka jako pro získávání zdrojů. Byl implementován registr továren ovladačů `ContentHandlerRegistry`, který spravuje továrny ovladačů pro různé typy obsahů a získává instance ovladačů v závislosti na požadovaném MIME typu obsahu.



Obrázek 5.7: Diagram tříd registru továren ovladačů

Třída ovladače, která zajišťuje zpracovávání daného typu obsahu, musí být konkrétní třídou nad abstraktní třídou `ContentHandler`. Pro zpracování obsahu se volá veřejná metoda `perform()`, která přijímá zdroj ke zpracování. V současné době je implementován pouze jeden ovladač – na zpracovávání HTML dokumentů `HtmlDocumentHandler`, který vkládá do událostní smyčky úlohu, jež zajišťuje tvorbu dokumentu, přidání nového dokumentu do historie, její aktualizaci a v poslední řadě zahájení parsování dokumentu. V implementaci jsou také připraveny návrhy tříd ovladačů na zpracovávání obrázků, textových souborů, XML dokumentů, médií aj.

Z důvodu potřeby podobné funkčnosti v jiné části implementace – výběru skriptovacího enginu v závislosti na MIME typu skriptu (viz kapitola 5.2.2) byly implementovány generické třídy pro správu továren objektů jakéhokoliv typu. Abstraktní továrna, která obsluhuje získávání obsahů podporovaného MIME typu, je v implementaci reprezentována třídou `MimeContentFactoryBase`. Každá konkrétní továrna redefinuje metodu pro vytváření obsahu `getContent()` a metody pro zjištění podporovaných MIME typů. Rozlišujeme dva druhy MIME typů:

- **explicitně podporované** – typy, které předem můžeme určit a o kterých víme, že továrna bude tyto typy podporovat, tzn. známe celou jejich podobu, např. `text/html`. Pro zjištění těchto typů slouží metoda `getExplicitlySupportedMimeTypes()`;
- **implicitně podporované** – typy, kde rozhodne o jejich platnosti až jejich formát, částečná shoda, např. pouze prefix `text/`; Pro test, zda se jedná o implicitně podporovaný typ je vyčleněná metoda `isImplicitlySupported()`.

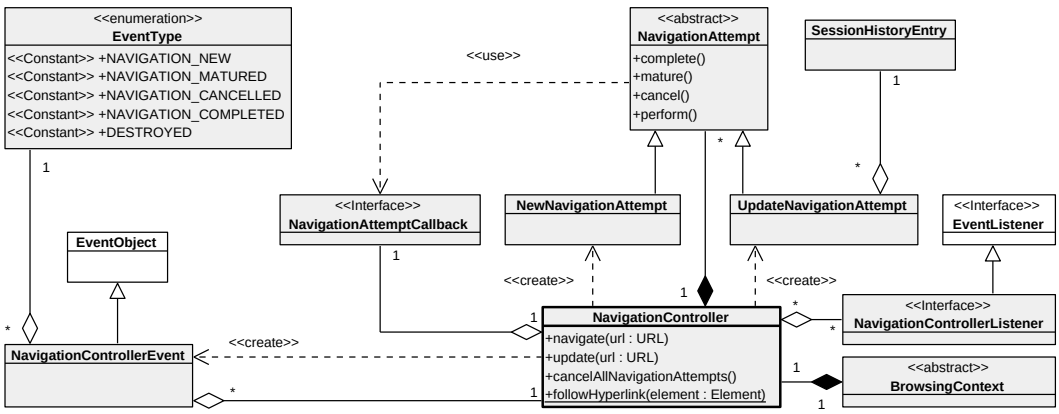
5.1.5 Navigace mezi dokumenty

V kapitole 5.1.3 byla popsána metodika, jak se získávají zdroje, jež lze využít nejenom pro stahování dokumentů, ale i pro získávání souborů kaskádových stylů nebo externích skriptů během načítání stránky. Předchozí kapitola 5.1.4 se zabývala výhradně zpracováním obsahu zdroje, vytvářením dokumentu a jeho zobrazením. Celý proces, který zahrnuje stažení zdroje a následné zobrazení dokumentu, přičemž je změněn předchozí aktivní dokument, lze nazvat navigací mezi dokumenty.

Požadavky na navigování nového dokumentu mohou přicházet asynchronně. V implementaci je vstupním bodem pro příchod nových požadavků navigační kontroler – třída `NavigationController`. Kontroler všechny požadavky registruje a pokouší se je obsloužit. Obsluha požadavků je delegována třídě požadavku `NavigationAttempt`, která o aktuálním stavu zpracování požadavku informuje svůj kontroler pomocí callback metod rozhraní `NavigationAttemptCallback`. Callback metody informují o následujících stavech:

- **vyzrání** – navigovaný dokument byl přidán do historie, ale stále není kompletní;
- **zrušení** – navigace byla zrušena uživatelem nebo kvůli nějaké chybě;
- **dokončení** – navigace byla kompletně provedena;
- **výběru kontextu** – byl vybrán efektivního procházecí kontext pro navigaci.

Navigace přechází do jednotlivých stavů zavoláním korespondujících metod navigace: `complete()`, `cancel()` nebo `mature()`. V závislosti na tom, jaká callback metoda byla navigačním požadavkem zavolána, je v navigačním kontroleru provedena k tomu odpovídající akce. Při výběru cílového kontextu je požadavek zaregistrován v kontroleru, naopak při zrušení nebo dokončení požadavku je požadavek zcela odregistrován. Jelikož bylo žádoucí informovat o stavech provádění požadavků uvnitř navigačního kontroleru i vnější okolí, byl vytvořen opět jednoduchý observer nad navigačním kontrolerem. Události, které mohou být vyvolány, kopírují metody callback volání z navigačního požadavku.



Obrázek 5.8: Diagram tříd navigačního kontroleru

Navigační kontroler vytváří celkem dva typy navigací. Prvním typem navigace je navigace `NewNavigationAttempt`, která vzniká po zavolání metody `navigate()` a která přidává do historie nový navigovaný dokument. Druhý typ navigace `UpdateNavigationAttempt` vznikající zavoláním metody `update()` přepíše dokument předaného záznamu historie, který je tímto aktualizován.

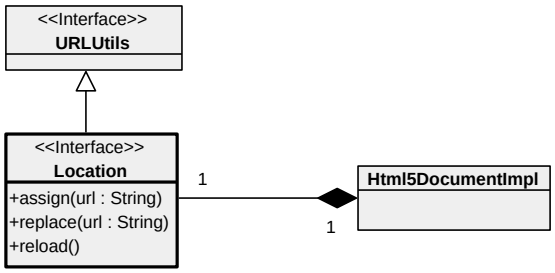
K vytvoření navigačního požadavku je zapotřebí nastavit:

- `navigationController` – navigační kontroler, který vlastní tento požadavek;
- `sourceBrowsingContext` – cílový procházecí kontext, kde navigace bude probíhat;
- `url` – adresu navigovaného zdroje;
- `exceptionEnabled` – nastaveno pokud se mají generovat DOM výjimky;
- `explicitSelfNavigationOverride` – když je nastaveno, bude cílovým procházecím kontextem zvolen procházecí kontext navigačního kontroleru a nebude se vyhodnocovat efektivní procházecí kontext;
- `replacementEnabled` – nastaveno pokud žádáme, aby byl přepsán právě aktivní záznam historie a smazány všechny jeho následující záznamy.

Pro vykonání a spuštění navigačního algoritmu je vyčleněná metoda `perform()`. Algoritmus provádí zjednodušeně následující body:

- **bezpečnostní kontroly** – testuje se, zda lze navigovat cílový procházecí kontext;
- **výběr efektivního cílového procházecího kontextu** – pokud není navigace prováděna v procházecím kontextu `<iframe>` se `seamless` atributem elementu, pak je efektivní cílový kontext vždy totožný s vybraným cílovým procházecícm kontextem;
- **navigace na fragment dokumentu** – když navigujeme pouze fragment dokumentu, pak se provede okamžitá navigace na pozici fragmentu, aktualizuje se historie a tímto bodem navigace ihned končí;
- **uvolnění dokumentu** – aktivní dokument cílového kontextu je uvolněn;
- **získání ovladače na stažení zdroje** – při úspěchu přechází navigační požadavek do asynchronního zpracovávání;
- **přesměrování** – když je zapotřebí přesměrovat na jiný zdroj;
- **získání ovladače** – vyhledání ovladače v registru `ContentHandlerRegistry`;
- **zpracování zdroje** – zavolání metody `process()` nad vyhledaným ovladačem.

Výše popsané třídy navigování se týkaly jádra uživatelského agenta. Navigovat dokumenty je ovšem přístupné i z klientského JavaScriptu. Pro tento účel slouží veřejné rozhraní `Location`, které bylo také zcela implementováno (obrázek 5.9). Každý dokument procházecího kontextu vlastní unikátní instanci tohoto rozhraní.

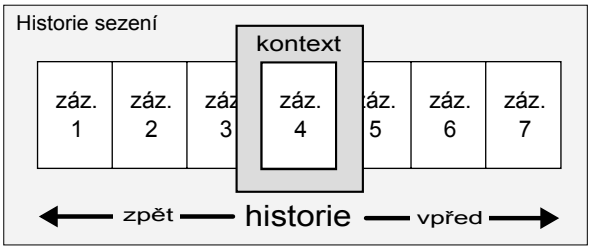


Obrázek 5.9: Diagram tříd rozhraní `Location`

Pro implementaci bylo využito navigačního kontroleru. Metody `assign()` a `reload()` volají metodu kontroleru `navigate()` s nastaveným příznakem `replacementEnabled`. Metoda `assign()` naviguje nový dokument bez přepsání právě aktivního dokumentu, příznak `replacementEnabled` není nastaven. Všechny metody nastavují `exceptionEnabled`, takže jsou navigačním požadavkem generovány bezpečnostní výjimky.

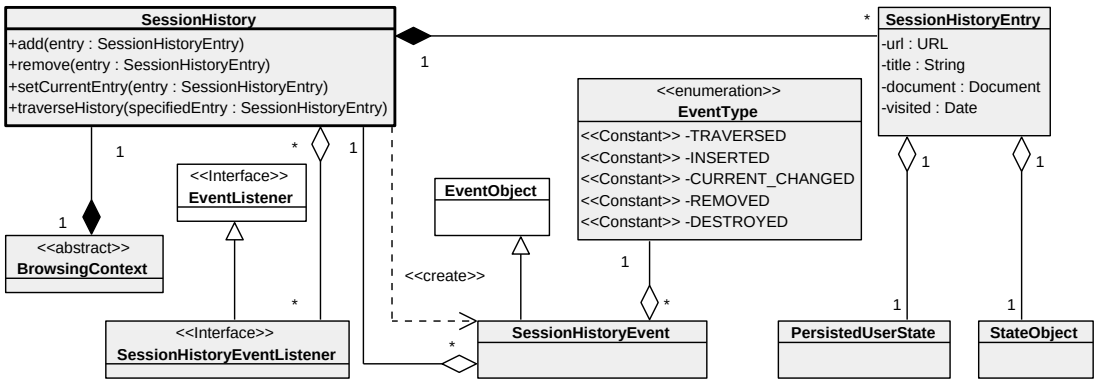
5.1.6 Procházení historie

Během navigování stránek dochází ke změně aktivních dokumentů. Všechny předchozí navigované dokumenty jsou nadále uchovávány v historii sezení. Průchodem historie sezení jde kdykoliv navracet dokumenty, které nejsou momentálně aktivní v současném procházecím kontextu. Historie sezení se vytváří v rámci konstrukce procházecího kontextu, jenž danou historii vlastní. Historii sezení si lze představit jako pole všech někdy navigovaných dokumentů s ukazatelem na aktivní dokument procházecího kontextu (obrázek 5.10).



Obrázek 5.10: Vizualizace historie sezení se záznamy mezi kterými lze posouvat

V implementaci je historie sezení představována třídou `SessionHistory` (obrázek 5.11). Historie je implementována jako lineární list záznamů historie `SessionHistoryEntry`. Záznam historie musí minimálně uchovávat adresu zdroje, odkud byl dokument získán. Dále by měl záznam obsahovat dokument, který byl vytvořen během navigace, jež vytvořila tento záznam. A dodatečně může záznam uchovávat titulek a datum navštívení stránky, stavový objekt `StateObject` a uživatelský kontext `UserPersistedState`. Stavovým objektem rozumíme objekty, které lze v historii uchovávat přes rozhraní `History`, popsané níže. Uživatelským kontextem rozumíme prostor pro ukládání např. pozic posuvníků před přesunutím na jiný záznam historie a jiné nastavení specifické pro danou HTML stránku.



Obrázek 5.11: Diagram tříd historie sezení

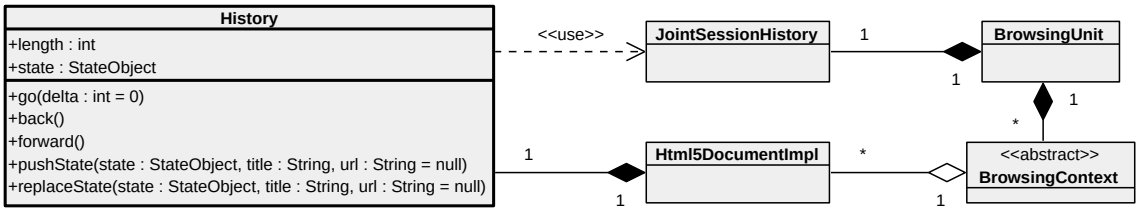
Tak jako u procházecích kontextů a navigačního kontroleru, implementuje objekt historie sezení jednoduchý observer mechanismus. Typy událostí generované historií sezení jsou:

- `TRAVERSED` – událost generovaná po dokončení přechodu z jednoho záznamu historie na nový záznam historie;
- `INSERTED` – vložení nového záznamu na konec listu historie sezení;
- `CURRENT_CHANGED` – pokud byl nastaven nový aktuální záznam historie. Událost zároveň značí, že došlo ke změně aktivního dokumentu procházecího kontextu, který vlastní dané historie sezení;
- `REMOVED` – smazání záznamu z historie sezení;
- `DESTROYED` – zrušení historie sezení.

Třída sezení historie `SessionHistory` obsahuje metody pro přidávání záznamů historie, jejich odebírání, filtrování a nastavování aktivního záznamu. Celý proces průchodu historií a změny aktivního záznamu na jiný záznam historie je obsáhlejší proces, který zahrnuje více kroků. Průchod historií sezení je implementován metodou `traverseHistory()`, jež bere jako parametr záznam sezení historie, který by měl být zvolen jako aktivní. Během průchodu historií se zjednodušeně provádí následující kroky:

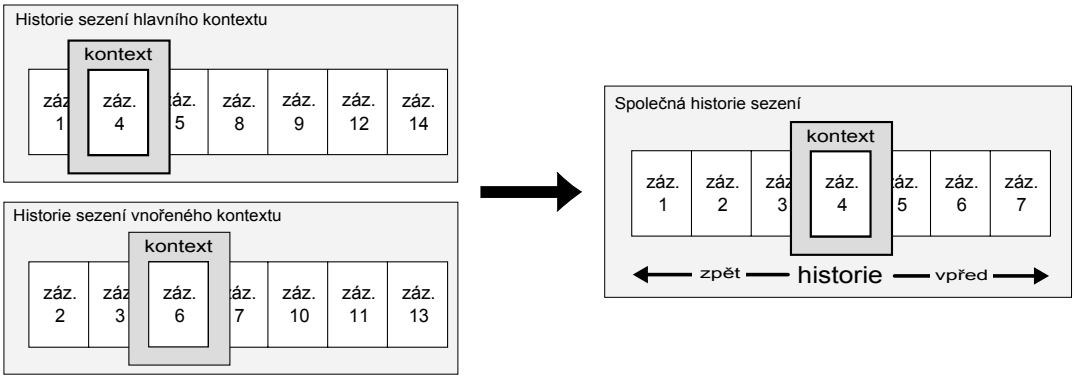
1. Zjištění, zda navigovaný záznam obsahuje objekt dokumentu. Pokud není dokument obsažen a my si přejeme posunout na tuto pozici v historii, musí se dokument obnovit zavoláním metody `update()` daného navigačního kontroleru.
2. Aktualizace uživatelského kontextu – uchování např. pozic posuvníků.
3. Odstranění všech úloh, které by byly po průchodu historií neplatné.
4. Nastavení nového aktivního záznamu historie sezení.
5. Generování události `load`, pokud je dokument kompletní.
6. Smazání všech záznamů před aktivním záznamem, pokud je to požadováno.
7. Aktualizace uživatelského prostředí z uchovaného uživatelského kontextu.
8. Kontrola, zda se změnil v URL fragment nebo stavový objekt a generování patřičných událostí `hashchange` nebo `popstate`.

Pro zpřístupnění procházení historie z kódu klientského JavaScriptu bylo nutné implementovat rozhraní `History` (5.12).



Obrázek 5.12: Diagram tříd rozhraní `History`

K implementaci rozhraní `History` bylo zapotřebí sjednotit všechny historie sezení všech procházecích kontextů procházecí jednotky. Rozhraní `History` totiž poskytuje procházení všech dokumentů, které kdy byly navigovány v procházecí jednotce. Spojení všech záznamů historií sezení dané procházecí jednotky proběhlo na základě času jednotlivých záznamů, kdy byly tyto záznamy vytvořeny, neboli kdy byly dokumenty záznamů navigovány. Aktuální záznam společné historie je vždy určen podle posledně vybraného záznamu během průchodu historií sezení v některém procházecím kontextu procházecí jednotky. Tvorba společné historie sezení je znázorněna na obrázku 5.13. Čísla záznamu reprezentují unikátní čísla generovaná v čase, kdy byl záznam vytvořen. Posledně vybraný záznam je záznam hlavního procházecího kontextu s číslem 4.

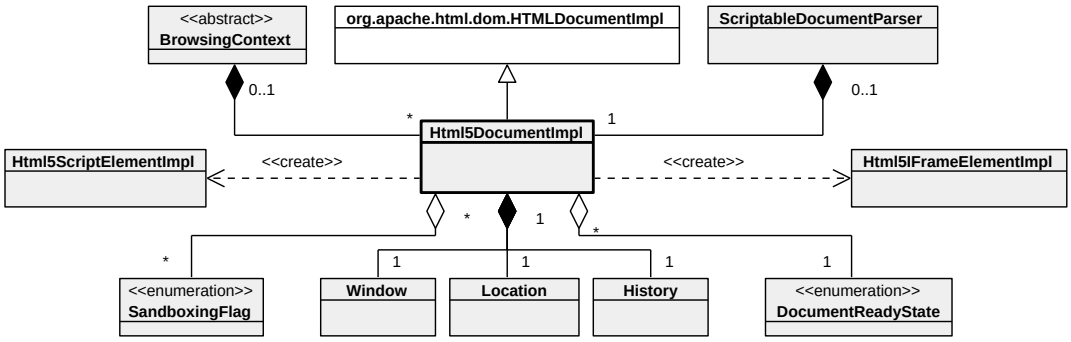


Obrázek 5.13: Vizualizace vytvoření sjednocené historie sezení

Sjednocenou historii sezení vlastní procházecí jednotka. V implementaci je představována třídou `JointSessionHistory`. Třída vnitřně registruje `BrowsingContextListener` a `SessionHistoryListener` nad všemi procházecími kontexty a historiemi sezení procházecí jednotky. Naslouchání odpovídajících událostí získává neustálé informace o změnách historie, čímž se může sama sebe aktualizovat – vytvářet sjednocenou historii. Reference na implementaci rozhraní `History`, jež zpřístupňuje sjednocenou historii z kódu JavaScriptu, se nachází v dokumentu. Kromě procházení společné historie sezení, rozhraní `History` umožňuje uchovávat stavový objekt `state`. Nový stavový objekt můžeme uložit pomocí speciální metody `pushState()`, která daný objekt nastavuje v aktuálním záznamu historie sezení. Když procházíme historií, je tento stavový objekt vždy z vybraného záznamu historie přečten a nastaven do atributu `state` rozhraní `History`.

5.1.7 Dokument a jeho parsování

Ačkoliv implementace rozhraní dokumentu viditelného z klientského JavaScriptu nebyla cílem této práce, bylo zapotřebí implementovat jeho vnitřní reprezentaci v kontextu jádra uživatelského agenta. V dokumentu jsou totiž uchovávány reference na implementovaná rozhraní `Window`, `Location` a `History`. Zároveň bylo nutno v dokumentu definovat vytváření elementů `<iframe>` a `<script>` tak, aby jejich implementace odpovídala specifikaci HTML 5. Třída, která reprezentuje implementaci dokumentu je nazvána `Html5DocumentImpl` a znázorněna na obrázku 5.14.



Obrázek 5.14: Diagram tříd dokumentu

Dokument dědí z třídy `HTMLDocumentImpl` knihovny Xerces a může být asociován s procházecím kontextem nebo parserem, který daný dokument vytvořil. Všechny procházené dokumenty mohou mít přidělený procházecí kontext. Dokument, který byl vytvořen zavoláním metody `createDocument()` v rámci provádění skriptu, žádnou referenci na procházecí kontext nevlastní. Stav dokumentu, ve kterých se může nacházet, je dán atributem výčtu `DocumentReadyState`:

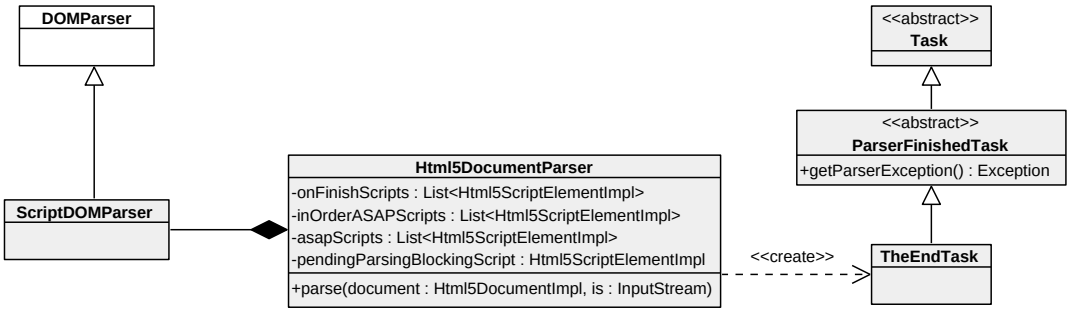
- `LOADING` – dokument je načítán parserem;
- `INTERACTIVE` – parser dokončil dokument, ale stále se čeká na některé zdroje;
- `COMPLETE` – dokument je zcela kompletně načten.

V dokumentu byly implementovány základní sandbox aspekty. Akce, které sandbox omezuje pro daný dokument, jsou uvedeny množinou sandbox přepínačů `SandboxingFlag`.

Během navigace nového dokumentu, jsou nové dokumenty vytvářeny v rámci zpracovávání obsahu zdroje (kapitola 5.1.4), konkrétně zavoláním statické metody nad dokumentem `createDocument()`. V rámci konstrukce nového objektu dokumentu vznikají instance rozhraní `Location` a `History`, které dokument nadále drží ve svém vlastnictví. Pokud byla metoda `createDocument()` zavolána s recyklovaným dokumentem `recycleDocument`, je pro nastavení reference objektu `Window` využit objekt `Window` recyklovaného dokumentu. Když není recyklovaný dokument nastaven, v rámci konstrukce se vytváří zcela nová instance objektu `Window`. Jakmile je nový dokument vytvořen, následuje jeho předání parseru, který zajistí jeho kompletní načtení.

Parser dokumentu představuje třída `ScriptableDocumentParser` (obrázek 5.15). Parsování se vyvolá zavoláním metody `parse()`, které se předá referenční objekt dokumentu a vstupní stream s dokumentem. Instance dokumentu není vytvářena parserem, dokument

je pouze parserem sestavován a nikoliv instanciován. Třída `ScriptableDocumentParser` reprezentuje vnější rozhraní pro komunikaci s parserem, přičemž parsování neimplementuje. Oddělení od parseru bylo zvoleno z důvodu případné budoucí změny parseru dokumentů na některý jiný, který by více vyhovoval specifikům HTML5. V současnosti je parsování zajištěno modifikovaným NekoHTML parserem `ScriptDOMParser`, který přidává podporu pro vytváření vnořených procházecích kontextů pro `<iframe>` elementy a spuštění skriptů v rámci parsování dokumentu, jak to vyplývá z HTML5 specifikace.



Obrázek 5.15: Diagram tříd parseru dokumentu

Podle specifikace by měl být parser inkrementální a reentrantní. Měla by být též implementována možnost jeho pozastavení, kde by byl kontext vrácen událostní smyčce na zpracování některé jiné úlohy. Pozastavení parseru je potřeba např. při dokončení parsování skriptu, který by měl být spuštěn v pořadí, a při blokování jiným nevykonaným skriptem. Skript může způsobit blokování dalšího skriptu, pokud nebyl stále spuštěn, např. z důvodu, že nebyl dosud stažen ze zdroje. V této situaci musíme parser pozastavit a počkat do té doby, dokud není zcela vykonán skript, který byl v pořadí před právě zpracovaným skriptem. Pozastavení parseru by se obvykle řešilo rotací událostní smyčky. Z důvodu nemožnosti rekonstrukce zásobníku volání NekoHTML parseru bylo pozastavení řešeno vyčleněním parseru do vlastního vlákna a jeho uspáním.

Třída `Html5DocumentParser` v současnosti uchovává referenci na skript, který bude blokovat parsování dokumentu, pokud bude zapotřebí spustit některý další skript v pořadí. V třídě jsou také listy skriptů:

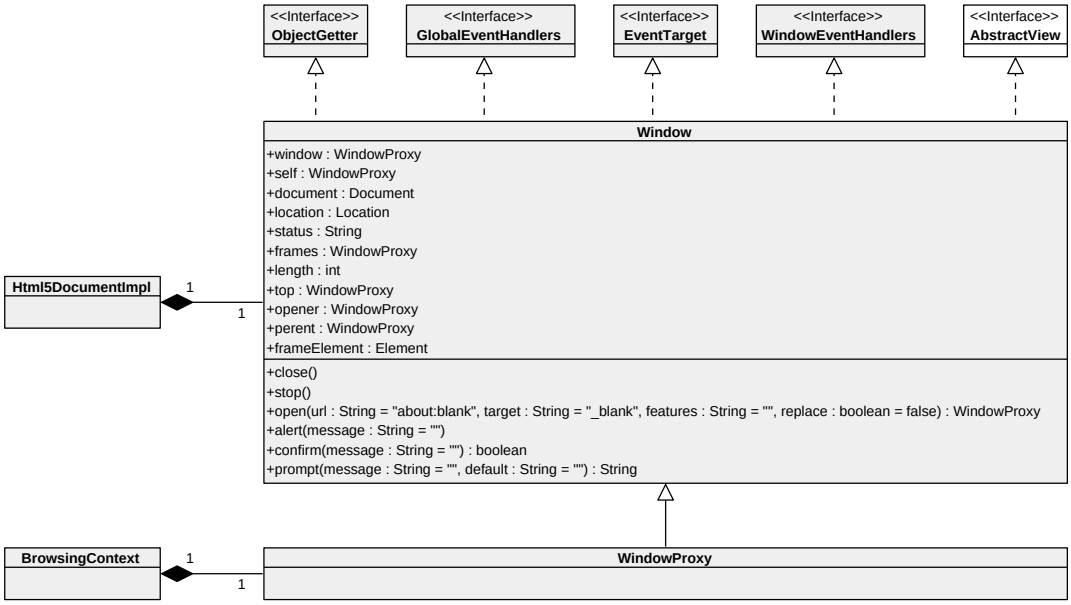
- `onFinishScripts` – skripty, které se mají spustit po dokončení parsování;
- `inOrderASAPScripts` – skripty k okamžitému spuštění jakmile budou dostupné ve správném pořadí, tak jak byly umístěny v DOM;
- `asapScripts` – skripty k okamžitému spuštění jakmile budou dostupné.

Když parser `ScriptDOMParser` dokončí parsování dokumentu, je vložena do událostní fronty úloha `TheEndTask`, čímž se navrátí řízení z asynchronního parsování opět událostní smyčce. Úloha nastaví stav dokumentu na `INTERACTIVE`, vykoná všechny skripty z listu `onFihishScripts`, vyvolá událost `DOMContentLoaded` nad parsovaným dokumentem a čeká, dokud nejsou vyprázdněny zbývající listy se skripty – `inOrderASAPScripts` a `asapScripts`. Po vykonání všech skriptů dojde k nastavení stavu dokumentu na `COMPLETE` a vyvolání události `load` nad daným dokumentem.

5.1.8 Rozhraní Window

V kapitole 5.1.7 jsme uvedli interní implementaci dokumentu, která obsahovala instance rozhraní `History` a `Location` – viditelných rohraní z klientského JavaScriptu. Zatím jsme ovšem nedefinovali, v rámci jakého rozhraní by měly být viděny právě tyto instance. V JavaScriptu pro zpřístupnění těchto rozhraní slouží globální objekt `Window`.

V implementaci je objekt `Window` implementován třídou `Window`. Instance nebo reference v případě recyklace objektu `Window` vzniká během konstrukce dokumentu. Implementovaná množina operací a atributů, s vynecháním atributů uživatelského rozhraní, je znázorněna na obrázku 5.16.



Obrázek 5.16: Diagram tříd rozhraní `Window`

Objekt `Window` využívá plně jádra uživatelského agenta, a tak jeho implementace je minimální. Atributy `window`, `self` a `frames` vrací objekt `WindowProxy` procházecího kontextu dokumentu vlastnícího objekt `Window`. Atribut `parent` vrací objekt `WindowProxy` rodičovského procházecího kontextu a atribut `top` objekt `WindowProxy` kontextu na nejvyšší úrovni. Jedná-li se o objekt `Window` vnořeného procházecího kontextu, vrací atribut `frameElement` element, přes který byl tento vnořený procházecí kontext vytvořen. Když je objekt `Window` vlastněn pomocným procházecím kontextem, atribut `opener` vrací objekt `WindowProxy` procházecího kontextu, který pomocný procházecí kontext otevřel. Atributy `location` a `history` přímo vrací aktuální instanci těchto rozhraní z dokumentu, který objekt `Window` vlastní. Metoda `close()` ruší procházecí kontext asociovaného dokumentu zavoláním metody `discard()`. Metoda `open()` naviguje nové dokumenty s využitím navigačního kontroleru `NavigationController` procházecího kontextu objektu `Window`. Metoda `stop()` ruší všechny požadavky na navigace. Poslední metody pro zobrazení uživatelských dialogů volají korespondující metody v asociovaném procházecím kontextu.

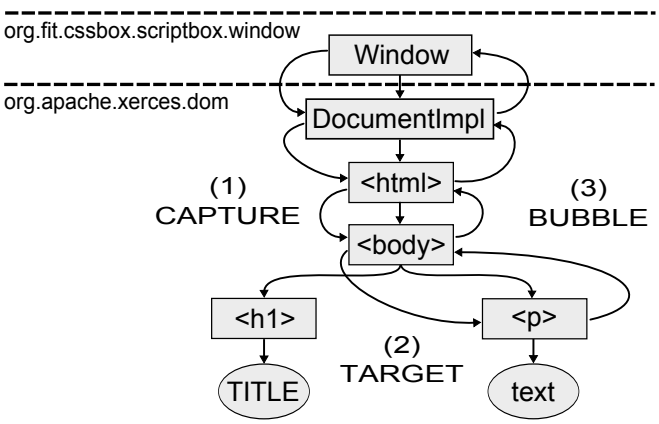
V objektu `Window` byly kompletně implementovány rozhraní `GlobalEventHandlers` a `WindowEventHandlers`, které přidávají podporu pro snadnou registraci obslužné rutiny pouhým přiřazením callback funkce do atributu bez voláni metody `addEventListner()`.

Objekt `Window` implementuje také rozhraní `ObjectGetter`, které slouží pro získávání objektů `WindowProxy` vnořených procházecích kontextů podle jejich jména nebo indexu. Indexem rozumíme pořadové číslo $0..n$, kde n počet všech vnořených procházecích kontextů určený položkou `length`.

V předchozím textu jsme se bavili několikrát o objektu `WindowProxy`. Tímto objektem myslíme speciální instanci, která má totožné chování jako aktuální globální objekt `Window` s tím rozdílem, že není asociována pevně s nastaveným dokumentem. Dokument objektu `WindowProxy` je proměnný v závislosti na tom, který je právě v daném procházecím kontextu objektu `WindowProxy` aktivní.

5.1.9 DOM události

Všechny uzly objektového modelu dokumentu by měly umožňovat generování událostí. Aby byla tato funkčnost splněna, musí uzly implementovat rozhraní `EventTarget`. Jelikož dokument `Html5DocumentImpl` dědí z třídy `DocumentImpl` knihovny Xerces, nebylo nutno generování událostí pro dokument řešit. Knihovna Xerces totiž rozhraní `EventTarget` implementuje pro všechny uzly dokumentu. V klientském JavaScriptu ovšem není kořenem DOM samotný dokument, jak je implementováno v Xerces, nýbrž objekt `Window`. Z tohoto důvodu bylo zapotřebí řešit provázání s objektem `Window` a generování událostí i do nového kořenového objektu, jak je znázorněno na obrázku 5.17.



Obrázek 5.17: Ilustrace toku události z objektu `Window` do objektu dokumentu

Události se generují zavoláním metody `dispatchEvent()` rozhraní `EventTarget` nad jakýmkoliv uzlem, který toto rozhraní implementuje. V implementaci Xerces jsou všechny události, které jsou předány metodě `dispatchEvent()`, dále delegovány chráněné metodě dokumentu `dispatchEvent()`, která požadované generování událostí implementuje. Jednotné místo v dokumentu pro generování událostí umožnilo snadnou změnu mechanismu generování událostí. Po provedené úpravě se dokument chová tak, jakoby byl kořenovým uzlem pro generování událostí objekt `Window`.

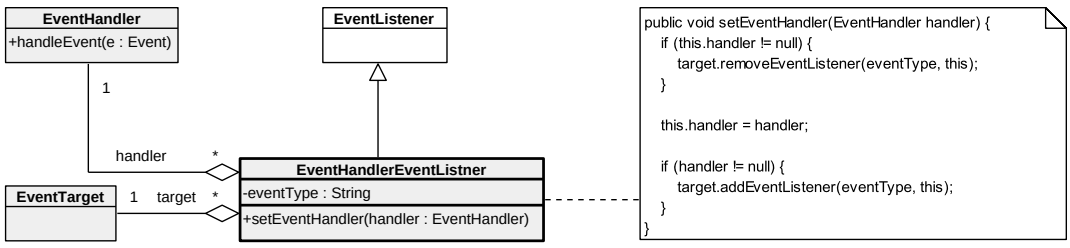
Během generování událostí v přizpůsobeném modelu mohou nyní nastat dvě situace. První situace je, pokud bylo zavoláno `dispatchEvent()` nad objektem `Window`. V tomto momentě se okamžitě volají všichni zaregistrovaní odběratelé dané události, přičemž fáze `CAPTURE` a `BUBBLE` se neprovádí, jelikož se jedná o kořenový uzel. Druhá situace již obnáší změnu zmíněné chráněné metody `dispatchEvent()`. Situace vzniká po zavolání metody

`dispatchEvent()` nad některým uzlem dokumentu. V tomto případě se provádí následující kroky:

1. Zavolání metody `dispatchEventFromDocument()` nad objektem `Window`, ve které se obslouží `CAPTURE` fáze.
2. Zavolání původní implementace `dispatchEvent()`, která zajistí propagaci události až do daného cíle a její návrat zpět do dokumentu.
3. Opětovné zavolání `dispatchEventFromDocument()` pro obsloužení `BUBBLE` fáze.

Elementy DOM, ale i objekt `Window` v klientském JavaScriptu neimplementují pouze rozhraní `EventTarget`, ale přidávají i vhodně pojmenované atributy, jejichž setter zajistí registraci nastavené událostní obslužné rutiny jako odběratele v daném objektu. Elementy implementují množinu atributů událostních obslužných rutin `GlobalEventHandlers`. Objekt `Window` implementuje dále rozhraní `WindowEventHandlers`.

Událostní obslužné rutiny musí v programu implementovat `EventHandler`, který jako rozhraní `EventListener` obsahuje pouze jednu metodu pro obsloužení dané události. Při nastavování obslužných rutin do korespondujících atributů událostních obslužných rutin rozhraní `GlobalEventHandlers` nebo `WindowEventHandlers` musela být řešena odregistrace odběratele staré obslužné rutiny a registrace nového odběratele nové obslužné rutiny. Za tímto účelem byla vytvořena třída `EventHandlerEventListener`, která registruje a odebírá událostní obslužné rutiny u cíle implementujícího `EventTarget` (obrázek 5.18).



Obrázek 5.18: Diagram tříd znázorňující registrování objektu `EventHandler`

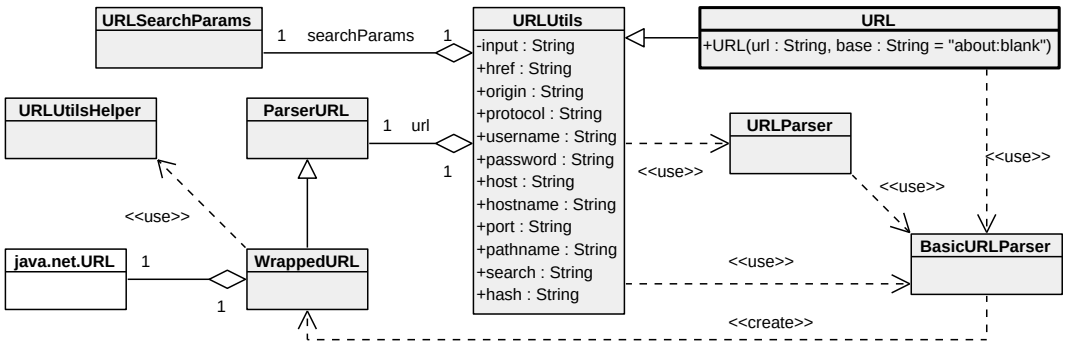
V současnosti jsou jádrem prohlížeče generovány události uvedené v následující tabulce 5.1.

Třída události	Popis třídy	Typy událostí
`Event`	jednoduché události neobsahující žádné další informace kromě názvu události	`load`, `beforescriptexecute`, `unload`, `afterscriptexecute`, `abort`, `DOMContentLoaded`
`PopStateEvent`	událost změny uloženého stavu ve společné historii sezení	`popstate`
`HashChangeEvent`	událost změny fragmentu v URL prohlížené stránky	`hashchange`
`ErrorEvent`	událost reprezentující chybu	`error`
`MouseEvent`	události vyvolané ukazatelem myši	`mouseover`, `mousedown`, `mouseup`, `click`, `dblclick`

Tabulka 5.1: Třídy podporovaných událostí

5.1.10 Rozhraní URLUtils a URL

Pro zkompletování výkladu všech rozhraní, která jsou v současné implementaci viditelná z klientského JavaScriptu nám chybí uvést poslední rozhraní `URL`, jež čerpá z dalšího rozhraní `URLUtils`. Rozhraní `URLUtils` není použité pouze pro objekty `URL`, ale i pro objekt implementující rozhraní `Location`. Ve své implementaci `URLUtils` rozhraní obaluje vstupní řetězec s URL adresou – `input`, který dále vnitřně parsuje za pomocí parseru `URLParser`, čímž vzniká objekt `url` třídy `ParserURL` (obrázek 5.19). Zpracovaná adresa `ParserURL` poskytuje náležitými gettery možnost získávání jednotlivých komponent URL adresy, které jsou přístupné právě přes rozhraní `URLUtils`.



Obrázek 5.19: Diagram tříd rozhraní `URL`

Parser `ParserURL` vnitřně používá `BasicURLParser`, který zajišťuje náležité parsování předaného URL řetězce. Jelikož tvorba parseru nebyla cílem této práce, `BasicURLParser` nevytváří nový objekt `ParserURL` podle referenčního algoritmu tak, jak bylo uvedeno ve specifikaci [22]. `BasicURLParser` pouze využívá již implementovaného parsovacího mechanismu v balíku `java.net` a třídy `URL`, jejíž objekt parsováním vznikne. Objekt třídy `URL` se obaluje do rozhraní `ParserURL`, což je zajištěno adaptující třídou `WrappedURL`.

Rozhraní `URL` vůči rozhraní `URLUtils` nepřidává žádnou další funkcionalitu, ale pouze definuje konstruktor pro tvorbu URL adresy, kterou objekt zapouzdřuje.

5.2 Scriptovací architektura pro dokumenty

Do HTML dokumentů mohou být vkládány klientské skripty různých skriptovacích jazyků dle rozebrané problematiky v návrhu – kapitole 4. Abychom umožnili snadnou implementaci klientských skriptovacích enginů pro různé skriptovací jazyky, bylo nutné vytvořit základní abstraktní architekturu, která bude pro všechny skriptovací enginy společná.

V této kapitole popíšeme implementaci abstraktního skriptovacího enginu a architekturu vytváření nativních objektů Javy tak, aby mohly být exportovány do skriptů. Fundamentální skriptovací architektura se nachází v balíku `org.fit.cssbox.scriptbox.script`.

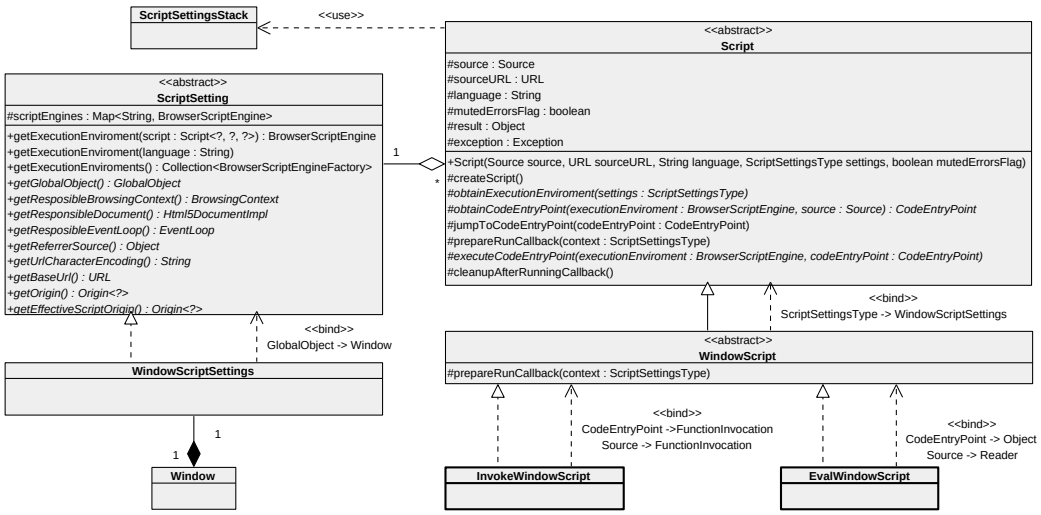
Architektura pro skriptací enginy vychází z kapitoly návrhu. Architektura byla v některých problematických částech mírně upravena tak, aby věrohodně kopírovala specifikaci HTML 5. Nejzásadnější změnou vůči návrhu je způsob spouštění skriptů. Skripty nejsou spouštěny pomocí jednotné třídy `DocumentScriptEngine`, která eviduje všechny skriptovací enginy, asociuje je pro dané dokumenty a implementuje veškerou funkcionalitu pro jejich spouštění. Z důvodu prevence proti případné nekompatibilitě s danou specifikací, umož-

není její snadnější a přesnější interpretace, bylo implementováno decentralizované spouštění skriptů. Odkaz na skriptovací engine nyní není vlastněn globální instancí spravující třídy `DocumentScriptEngine`, ale dokumentem samotným v objektu `ScriptSettings`.

Implementované řešení provádění skriptů v dokumentu a objekt `ScriptSettings` je popsán v kapitole 5.2.1. Injekce objektů do hlavního scope (kapitola 4.3), jež byla implementována bez odlišností od návrhu, je rozebrána v kapitole 5.2.5. Tvorba JavaScriptového enginu podle JSR 223 (kapitola 4.4) byla též kompletně dodržena. Rhino engine byl v mnoha ohledech přizpůsoben a vylepšen tak, aby ho bylo možné využít pro klientské skripty. O implementovaném řešení JavaScript enginu pojednává kapitola 5.2.2.

5.2.1 Rozhraní skriptů dokumentu

Podle specifikace HTML5 bylo zapotřebí implementovat společné rozhraní pro skripty dokumentu. Rozhraní pro skripty je v implementaci reprezentováno abstraktní třídou `Script`. Každý skript využívá pro svoji tvorbu a běh určitého nastavení daného abstraktní třídou `ScriptSettings`.



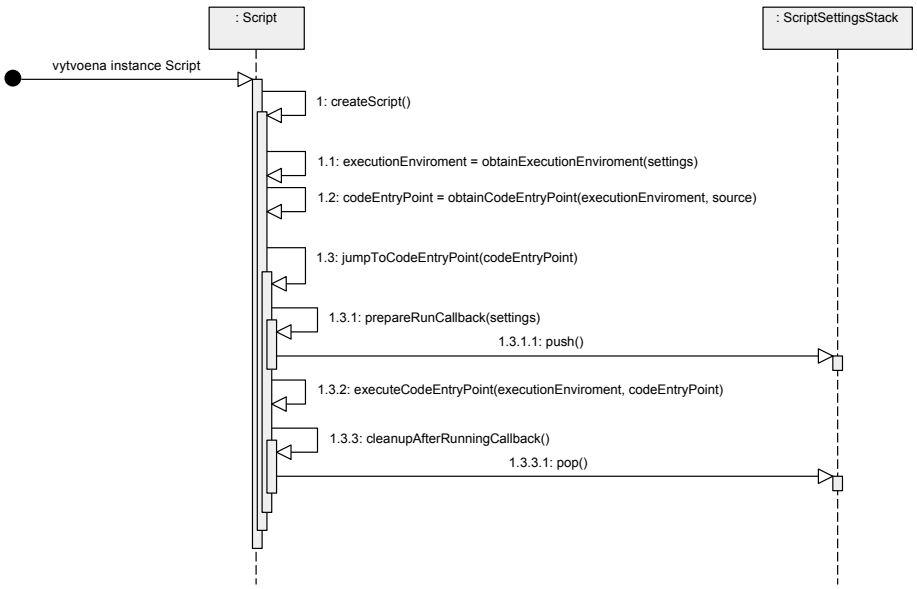
Obrázek 5.20: Diagram tříd abstraktního rozhraní skriptu

Nastavení pro skripty deklaruje důležité aspekty prostředí, ve kterém by se měl skript vykonat. Třída `ScriptSettings` je abstraktní a v současné implementaci ji realizuje jediná třída `WindowScriptSettings`. Instance nastavení je vytvářena vždy během konstrukce objektu `Window`, který je zároveň globálním objektem pro JavaScriptový engine. Třída skriptu `WindowScriptEngine` poskytuje referenci zejména na:

- dokument `Html5DocumentImpl`, ve kterém se nachází spuštěný skript;
- globální objekt `Window`;
- bázovou URL adresu, která bude použita pro vytváření URL adres z adres relativních;
- událostní smyčku `EventLoop` určenou pro skripty;
- původ skriptu `Origin`, jeho kódování, atd.

Jelikož nastavení pro skripty je unikátní pro každý objekt Window, který je přímo v 1 : 1 asociaci s objektem dokumentu, bylo nastavení využito pro tvorbu a uchovávání skriptovacích enginů dokumentu. Skriptovací enginy lze vytvářet zavoláním metody nastavení skriptu `getExecutionEnviroment()`. Jakmile je skriptovací engine již jednou vytvořen, je v konkrétním nastavení uchován a během příštího dotazu opět navrácen. Při vytváření neustále nových skriptovacích enginů by docházelo ke ztrátě uchovaného kontextu uvnitř enginů.

Abstraktní třída pro skripty `Script` potřebuje pro svoji konstrukci zdroj skriptu `Source`, jazyk skriptu, URL ze které byl případně skript získán a nastavení pro skripty generického typu `ScriptSettingsType`. V rámci tvorby instance skriptu se skript provede zavoláním chráněné metody `createScript()`. Způsob spuštění skriptu je znázorněn na obrázku 5.21. Jakmile se ověří, zda je skriptování povoleno, dojde nejprve k získání skriptovacího enginu metodou `obtainExecutionEnviroment()`. Pak dojde k přípravě skriptu a získání vstupního bodu k vykonání `CondeEntryPoint` metodou `obtainCodeEntryPoint()`, který je vykonán metodou `jumpToCodeEntryPoint()`. Vykonávací metoda před spuštěním skriptu vkládá na zásobník `ScriptSettingsStack` aktuální nastavení skriptu, jež je po skončení vykonávání opět ze zásobníku odstraněno. Zásobník skriptů je potřebný v některých situacích, např. když chceme znát některé základní informace o nastavení skriptu, který spustil některou funkci jádra uživatelského agenta. Během navigování stránek metodou `open()` objektu `Window` se využívá zásobníku k určení zdrojového procházecího kontextu, který je totožný s responsivním procházecím kontextem skriptu.



Obrázek 5.21: Sekvenční diagram znázorňující tvorbu skriptu

Jedinými konkrétními třídami abstraktní třídy `Script` jsou třídy `EvalWindowScript` a `InvokeWindowScript`. Obě zmíněné třídy rozšiřují třídu `WindowScript`, jež předefinovává metodu `prepareRunCallback()`, do které přidává test, zda je dokument objektu `Window` plně aktivní. Pokud není dokument aktivní, vrací metoda nepravdu a skript není spuštěn.

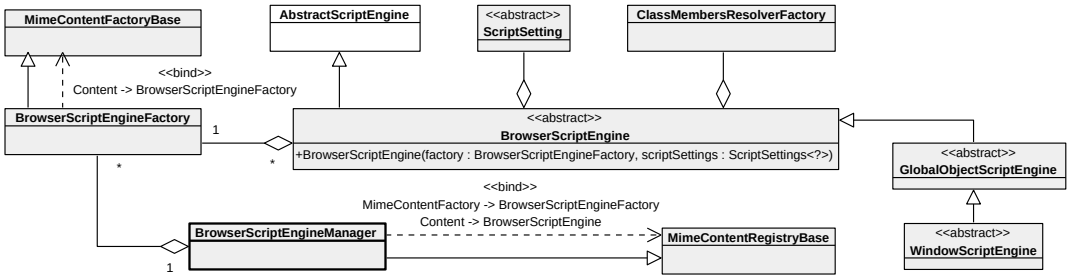
Třída `EvalWindowScript` slouží k prostému vykonávání spustitelného kódu v nalezeném skriptovacím enginu. Zdrojem pro skripty `EvalWindowScript` je `Reader`. Během tvorby vstupního bodu `CodeEntryPoint` se vrací buď předaný zdroj skriptu, tzn. `Reader`, nebo zkompilovaný zdroj `CompiledScript`. Při vykonávání skriptu se volá buď `eval()` skripto-

vacího enginu nebo `eval()` zkompilovaného skriptu v závislosti na tom, jaký vstupní bod byl metodě `executeCodeEntryPoint()` předán.

Třída `InvokeWindowScript` plní funkci přímého volání funkcí v kontextu skriptovacího enginu. Zdrojem skriptu je rozhraní `FunctionInvocation`, které informuje o názvu funkce a argumentech, se kterými byla zavolána. Metoda vykonávání skriptu volá metodu `invoke()` vybraného skriptacího enginu s parametry, které byly získány pomocí rozhraní `FunctionInvocation`.

5.2.2 Klientské skriptovací enginy

Abstraktní skriptovací engine pro provádění skriptů popsaných v předchozí kapitole byl implementován podle návrhu tak, aby byl kompatibilní se specifikací JSR 223. Abstraktní engine dědí z třídy `AbstractScriptEngine` určené pro všechny skriptovací enginy standardního skriptovacího API. V implementaci je abstraktní engine uživatelského agenta reprezentován třídou `BrowserScriptEngine` (obrázek 5.22).



Obrázek 5.22: Diagram tříd abstraktního skriptovacího enginu

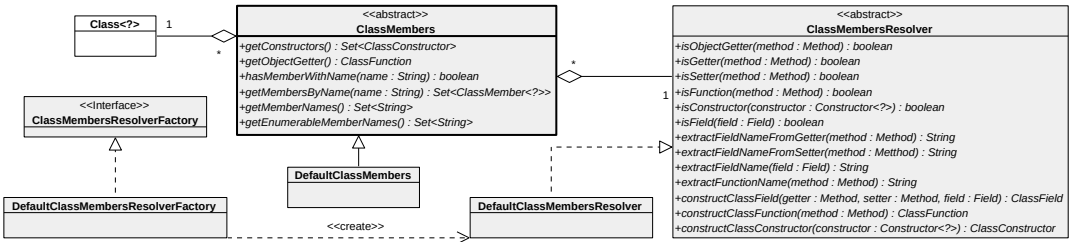
Všechny skriptovací enginy jsou vytvářeny pro některý konkrétní dokument, nad nímž je skriptovacím enginem uchováván skriptovací kontext. Pro jaký dokument byl skriptovací engine vytvořen, lze zjistit z nastavení pro skripty `ScriptSettings`, které je předáváno abstraktnímu skriptovacímu enginu během jeho konstrukce. V rámci třídy abstraktního skriptovacího enginu se s objektem `ScriptSettings` přímo nepracuje, ale v dalších specifičtějších třídách je dále využíván pro zjišťování globálního objektu. Globální objekt bývá obvykle konkrétními skriptovacími enginy implementován do enginového scope daného skriptovacího enginu. Konkrétně třída `WindowScriptEngine` reprezentuje rozhraní pro zabudování globálního objektu `Window`.

Každý skriptovací engine `BrowserScriptEngine` obsahuje referenci na továrnu resolverů členů tříd `ClassMembersResolverFactory`. Resolver je použit k určení viditelných členů exportovaných objektů Javy a pro implementaci globálního objektu. Resolver je blíže popsán v následující kapitole 5.2.3.

Skriptovací enginy jsou vytvářeny továrnami `BrowserScriptEngineFactory`, které jsou evidovány a registrovány v singleton třídě `BrowserScriptEngineManager`. Manažerská třída spravuje všechny továrny skriptovacích enginů. Třída poskytuje na dotázání továrny vnějšímu okolí nebo případně ihned vytváří konkrétní skriptovací enginy některé továrny. Třídy `BrowserScriptEngineFactory` a `BrowserScriptEngineManager` rozšiřují třídy pro spravování továren obsahů v závislosti na MIME typu obsahu, které byly blíže popsány v kapitole 5.1.4. Rozhodujícím faktorem pro vybrání odpovídající továrny skriptovacího enginu je MIME typ skriptu, na jehož základě je v manažeru továren vyhledáváno.

5.2.3 Koncepty exportu objektů Javy

Z důvodu snadného přidávání nové implementace klientských skriptovacích enginů byly implementovány koncepty exportu nativních objektů Javy do kontextů skriptovacích enginů. Když chceme exportovat objekt, je výhodné vědět a určit, které prvky objektu mají být exportovány a jak by tyto prvky měly být reprezentovány v klientských skriptovacích enginech. Prvky zde myslíme členy třídy – atributy, metody nebo konstruktory třídy. Za účelem specifikace exportovatelných členů tříd vznikla podpora, která přesně vyčleňuje jednotlivé členy tříd. Exportovatelné členy jsou vyhodnocovány ve třídách rozšiřující abstraktní třídu `ClassMembers`. Výchozí a jediná implementace pro určování členů tříd je hotova ve třídě `DefaultClassMembers`. Abstraktní třída `ClassMembers` vlastní odkaz na třídu, jejíž členy jsou vyhodnocovány, a odkaz na resolver, který jednotlivé členy rozpoznává.

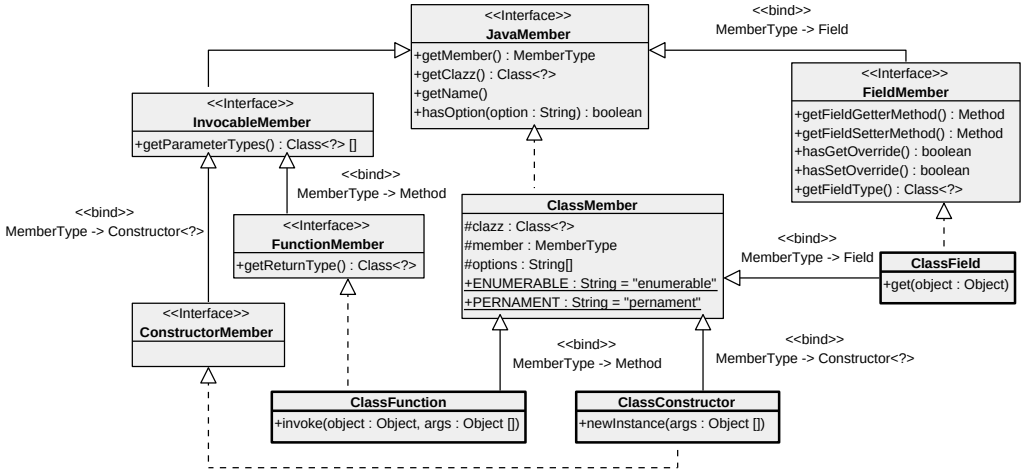


Obrázek 5.23: Diagram tříd znázorňující třídu reprezentující členy tříd

Třídy resolverů slouží náležitou implementací metod začínajících prefixem `is...()` k určení, zda má být Java člen exportován nebo nikoliv. Pokud je člen exportovatelný, pak resolvery zprostředkovávají i extrakci názvů členů, pod kterými jsou členy viditelné v klientských skriptech. Pro účel extrakce jsou vyčleněny metody začínající prefixem `extract...()`. Závěrem resolvery vytváří metodami s prefixem `construct...()` obaly exportovatelných standardních Java členů – členů `Constructor`, `Field` a `Method`. Poslední metodou poskytovanou resolvery, je metoda testu `isObjectGetter()` ověřující, zda je předaná metoda metodou pro indexované získávání objektů z objektu zkoumané třídy. Asociativní získávání objektů z objektu, neboli vytvoření asociativního pole z objektu, bylo implementováno z důvodu potřeby indexovaného získávání objektů `WindowProxy` vnořených procházecích kontextů z objektu `Window` podle jejich jména nebo indexu. Výchozí resolver `DefaultClassMembersResolver` exportuje všechny členy tříd Javy, přičemž jejich názvy extrahuje pomocí reflexe.

Pro rozšíření funkčnosti standardních Java členů byly vytvořeny speciální obalující třídy `ClassFunction`, `ClassConstructor` a `ClassField` tak, jak je znázorněno na obrázku 5.24. Obaly Java členů přidávají pomocné metody pro export a hlavně obsahují odkaz na pole možností `options`, které specifikuje způsob, jak by měl být člen implementován v klientském skriptu. V současné implementaci je možno uchovávat možnosti:

- `ENUMERABLE` – slouží pro určení, zda člen bude zúčastněn v seznamu všech výčtových členů objektu. V JavaScriptu umožňuje povolit nebo zabránit výpisu vlastnosti objektu, např. při průchodu `for (var property in object)`. Tato možnost reflektuje ECMAScript atribut `[[Enumerable]]` [20];
- `PERNAMENT` – zda člen bude natrvalo svázán s objektem, v němž je definován, nebo může být přepsán během vykonávání skriptu za uživatelský objekt. Tato možnost zjednodušeně reflektuje Web IDL atributy `[Unforgeable]` a `[Replaceable]` [24].

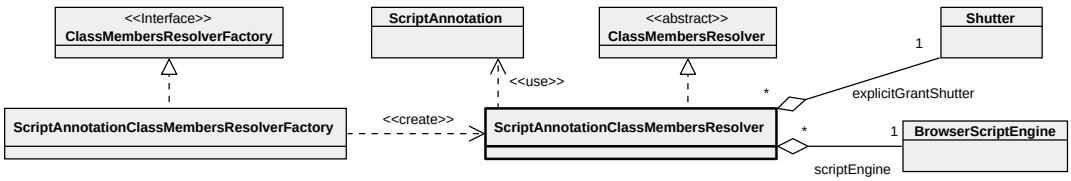


Obrázek 5.24: Diagram tříd členů tříd

Z důvodu uchovávání a svázání členů tříd s objekty, které byly exportovány, byly implementovány třídy členů objektů: `ObjectFunction`, `ObjectConstructor`, `ObjectField`. Tyto třídy, jako třídy členů tříd, implementují rozhraní `FunctionMember`, `ConstructorMember` a `FieldMember`. Strom dědičnosti tříd členů objektů je obdobný jako u členů tříd. Objekty členů navíc od objektů členů uchovávají instanci exportovaného objektu a implementačně se od tříd členů tříd příliš neliší, proto zde nebudou blíže popisovány.

5.2.4 Export objektů založený na anotacích

Systém exportu nativních objektů Javy využívající výchozí implementací resolveru členů třídy `DefaultClassMembersResolver`, popsaný v předchozí kapitole 5.2.3, umožňoval kompletní export objektů do klientských skriptů. Kompletní export není ovšem téměř nikdy žádoucí, jelikož do exportu jsou zahrnuty i např. metody třídy `Object`, ze které dědí každá třída Javy. Velmi často se prolíná vnitřní implementace členů objektu pro export a implementace členů, který by měly být ve skutečnosti exportovány. Nastala otázka, jak jednotlivé členy odlišit. Řešení se nabídlo v anotacích Javy, díky kterým bylo možné přesně určit exportovatelné členy a rozšířit exportování o další přídavné funkčnosti.



Obrázek 5.25: Diagram tříd resolveru členů tříd založeném na anotacích

Pro export založený na anotacích členů tříd byl vytvořen nový resolver členů implementovaný třídou `ScriptAnnotationClassMembersResolver`. Resolver pro svoji funkci potřebuje referenci na skriptovací engine, který provádí vyhodnocování členů tříd. Bez znalosti skriptovacího enginu bychom nemohli rozhodnout, zda je daný člen exportovatelný pro konkrétní skriptovací engine. Pokud by bylo žádoucí exportování stejných členů do všech skriptovacích enginů a nemít možnost volby skriptovacího enginu, pro který je export validní, pak by reference na skriptovací engine nemusela existovat. Třída resolveru obsahuje

ještě referenci na objekt `Shutter`, který zde neslouží k restrikci přístupu pro určité členy, třídy a balíčky, nýbrž k explicitnímu povolování členů, které nenesou žádnou anotaci. Pro implementaci funkčnosti třídy resolveru se využívá pomocné třídy `ScriptAnnotation`, ve které se nachází pouze statické metody. Metody pomáhají určit, zda je daný člen exportovatelný na základě anotací pro daný skriptovací engine.

V současné implementaci jsou podporovány skriptovací anotace pro členy třídy a třídy uvedené v tabulce 5.2.

Anotace	Působnost anotace	Popis funkce anotace
`@InvisibleField`	Atribut	zabránění exportu atributu
`@InvisibleFunction`	Metoda	zabránění exportu funkce
`@ScriptClass`	Třída	anotace pro exportovatelnou třídu – nemusí být uvedena
`@ScriptConstructor`	Konstruktor	exportovatelný konstruktor objektu
`@ScriptField`	Atribut	exportovatelný atribut
`@ScriptFunction`	Metoda	exportovatelná funkce
`@ScriptGetter`	Metoda	exportovatelný atribut na základě getter metody
`@ScriptSetter`	Metoda	exportovatelný atribut na základě setter metody

Tabulka 5.2: Podporované skriptovací anotace

Každá skriptovací anotace obsahuje některé atributy, které omezují a ovlivňují export členů tříd. Všechny anotace mají společný jeden atribut `engines` s polem skriptovacích enginů, pro které je daná anotace validní. Pokud je pole `engines` prázdné, pak se předpokládá, že je člen s touto anotací viditelný pro všechny skriptovací enginy.

Jedinou anotací, která není určena pro členy třídy, je anotace `@ScriptClass`, jež se uvádí nad typem třídy. Tato anotace umožňuje definovat rozsah automatického zahrnutí exportovatelných členů třídy, bez nutnosti definování odpovídající anotace u daného členu. Anotace přidává k popsanému atributu `engines` další atribut s polem `options`. Pokud zahrneme v poli s možnostmi anotace `options` např. hodnotu `ALL_METHODS`, tak budou exportovány všechny metody dané třídy, nad kterou je anotace uvedena. Další podporované možnosti anotace jsou: `ALL_FIELDS`, `ALL_STATIC_METHODS`, `ALL_STATIC_FIELDS`, `ALL_CONSTRUCTORS`. Jedinými anotacemi, které mohou ovlivnit automatické exporty po uvedení některé možnosti anotace `@ScriptClass`, jsou anotace exkluze `@InvisibleField` a `@InvisibleFunction`. Tyto anotace slouží k zabránění exportu členu třídy, který by byl normálně exportován.

Když žádáme vytvoření některého atributu v klientském skriptu, máme hned tři možnosti jak toho docílit. Pro definici atributu můžeme použít anotaci `@ScriptField`, která přímo zpřístupní atribut, nad kterým je anotace uvedena. Dále máme k dispozici anotace `@ScriptGetter` a `@ScriptSetter`, které vytvoří atribut z getter nebo setter metody, nad níž je anotace uvedena. Resolver pro anotace umožnuje pro všechny tři uvedené anotace automatické odvození názvu atributu. Název atributu lze předefinovat uvedením atributu anotace `field`, kde lze zvolit vlastní název pro atribut pro klientské skriptovací enginy. Pokud je jeden a tentýž atribut definován na více místech pomocí různých anotací, pak lze zvolit, zda by mělo mít přednost volání getteru nebo setter pro získávání atributu před přímým přístupem k atributu. Přednosti getterů a setterů se definují v možnostech anotací

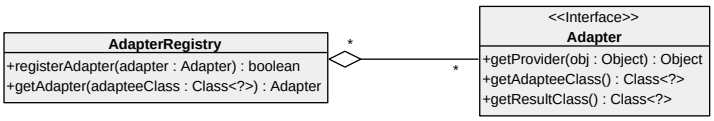
hodnotou `FIELD_GET_OVERRIDE` nebo `FIELD_SET_OVERRIDE`. Chceme-li zahrnout metody getterů a setterů do exportovatelných funkcí, pak lze v možnostech anotace uvést hodnotu `CALLABLE_GETTER` nebo `CALLABLE_SETTER`.

U všech výše uvedených anotací, které sloužily pro export členu třídy, lze také zvolit, zda by tento člen měl mít nastavenou vlastnost `ENUMERABLE` popsanou v kapitole 5.2.3. Vlastnost `PERNAMENT` nelze v současné implementaci nastavovat. Proto všechny členy exportované pomocí skriptovacích anotací mají možnost `PERNAMENT` pevně nastavenou.

5.2.5 Další implementovaná funkčnost

V kapitole 4.3 jsme navrhli injekci do hlavního scope. Koncept injekce byl kompletně dodržen tak, jak byl navržen. Jednotlivé injektory `ScriptContextInjector` po zavolání metody `registerScriptContextInject()` registrují automaticky nesoucí injekci u všech továren skriptovacích enginů, ke kterým injektory náleží. Přejeme-li si zaregistrovat injektor u všech zaregistrovaných továren skriptovacích enginů, tak stačí během konstrukce injektoru předat speciální konstantu `ALL_SCRIPT_ENGINE_FACTORIES`. V současnosti jediným implementovaným injektorem, který je společný pro všechny skriptovací enginy uživatelského agenta, je `URLInjector`, který vkládá objekt pro konstrukci URL do hlavního scope všech skriptovacích enginů. Pro JavaScriptový stroj byl dále implementován injektor vkládající speciální objekty, které reprezentují třídy klientského JavaScriptu, viz kapitola 5.3.2. Posledním implementovaným injektorem je injektor vkládající objekt konzole v rámci ukázkové aplikace.

Během implementace se vyskytla potřeba převádět objekty na různá rozhraní, nejčastěji rozhraní neobsahující skriptovací anotace na objekty, které jsou již anotované a tak přístupné pro klientské skripty. Pro tento účel byl implementován jednoduchý registr adaptérů umožňující požadovanou funkčnost (obrázek 5.26). Adaptéry jsou používány třeba pro převod Xerces implementace událostí `EventImpl`, `MouseEventImpl` na třídy `AdaptedEvent`, `AdaptedMouseEvent` za pomocí adapterů `EventAdapter` a `MouseEventAdapter`. Jednotlivé skriptovací enginy by měly využívat tento registr adaptérů při zviditelňování nativních objektů Javy v klientských skriptech.



Obrázek 5.26: Diagram tříd registru adaptérů

5.3 Klientský JavaScriptový engine

V minulých kapitolách jsme popsali obecné společné rozhraní pro všechny skriptovací enginy. V této kapitole se zaměříme na konkrétní implementaci jediného skriptovacího enginu, který je nyní podporován – skriptovacího enginu JavaScriptu.

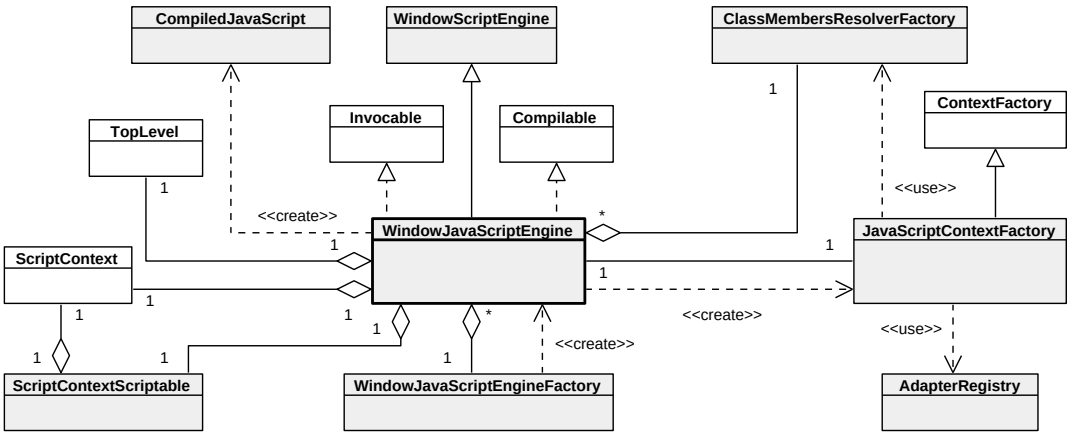
JavaScriptový skriptovací engine byl implementován s využitím knihovny Rhino, která poskytuje jádro interpretu JavaScriptu. Implementovaný skriptovací engine je kompatibilní se specifikací JSR 223. Implementace JavaScriptového enginu podle JSR 223 v JDK nebylo využito z důvodu přizpůsobení skriptovacího enginu a kompletní kontroly nad zabudovaným prostředím Rhino. Standardní skriptovací API pro engine bylo implementováno zcela od základu. Jakým způsobem proběhlo propojením s Rhinem, je uvedeno v kapitole 5.3.1.

V rámci konstrukce skriptovacího enginu dochází k tvorbě globálního scope. Proces, který implementuje atributy a metody globálního objektu do globálního scope, je popsán v kapitole 5.3.2.

Knihovna Rhino exportuje nativní objekty Javy do prostředí JavaScriptu tak, že je zapouzdřuje do objektů `NativeJavaObject`, které implementují rozhraní `Scriptable`. Pro specifikaci členů objektů, jež mají být implementovány, využívá Rhino třídy `JavaMembers`, která ovšem není veřejně přístupná. Z důvodu potřeby definice vlastních exportovatelných členů v rámci exportovaných objektů Javy, bylo nutné vytvořit vlastní zapouzdřující objekt. Jak zapouzdření proběhlo, je popsáno v kapitole 5.3.3.

5.3.1 Rozhraní JavaScriptového enginu

Skriptovací engine JavaScriptu je vytvářen továrnou `WindowJavaScriptEngineFactory`. V implementaci jej představuje třída `WindowJavaScriptEngine` (obrázek 5.27). Pro jedno vlákno může být vytvořen pouze jeden skriptovací engine. Omezení je dáno z důvodu, že je skriptovací engine přímo spjat s kontextem Rhina, který se vytváří pro aktuální běžící vlákno. Skriptovací kontext Rhina slouží pro běh skriptů a je vytvářen před spuštěním nového skriptu. K vytváření kontextu se využívá továrny `JavaScriptContextFactory`, která je instanciována v rámci konstrukce skriptovacího enginu. Během konstrukce enginu také dochází k vytvoření globálního scope `TopLevel`, jehož tvorba je blíže popsána v další kapitole 5.3.2.



Obrázek 5.27: Diagram tříd skriptovacího enginu JavaScriptu

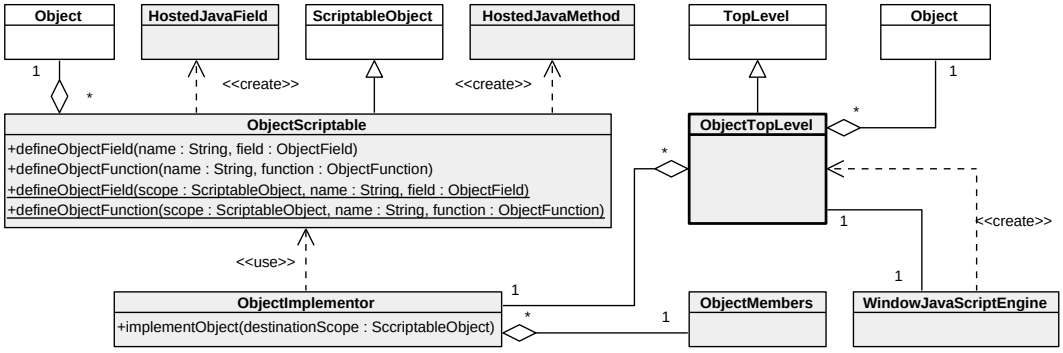
V rámci konstrukce továrny skriptovacího kontextu `JavaScriptContextFactory` se registrují jednotlivé podporované adaptéry pro skriptovací engine. Aktuálně jsou registrovány adaptéry: `EventAdapter`, `UIEventAdapter`, `MouseEventAdapter` a `ErrorAdapter`, jež převádí nativní objekty událostí na objekty viditelné ve skriptu. Když dochází k tvorbě kontextu, je pro kontext nastavena obalující továrna, která řetězí více metod obalování s využitím dekorátoru `WrapFactoryDecorator` blíže popsaného v kapitole 5.3.3.

Skriptovací engine vlastní referenci na továrnu resolverů členů tříd, která se využívá pro export nativních objektů Javy. V aktuální implementaci se využívá továrny založené na anotacích `ScriptAnnotationClassMembersResolverFactory`. Skriptovací engine umožňuje volat funkce pomocí rozhraní `Invocable` nebo kompilovat skripty s využitím rozhraní `Compilable` do objektů `CompiledJavaScript`.

Abychom mohli implementovat data binding podle specifikace JSR 223, bylo nutno vytvořit speciální scope – objekt `ScripContextScriptable`. Tento scope zpřístupňuje uložená data pomocí `Bindings` klientskému JavaScriptu a naopak z klientského JavaScriptu zprostředkovává rozhraní pro uložení dat skriptu.

5.3.2 Implementace globálního scope

V rámci konstrukce skriptovacího enginu dochází k tvorbě globálního scope. Globální scope je reprezentován v Rhinu třídou `TopLevel`. Jelikož klientský JavaScript obsahuje globální scope, kde je implementován globální objekt JavaScriptu `Window`, byla vytvořena rozšiřující třída `ObjectTopLevel` (obrázek 5.28), která daný objekt do globálního scope vkládá. Globální objekt `Window` je získáván skriptovacím enginem z nastavení skriptu `ScriptSettings` a dále předáván v rámci konstrukce třídě `ObjectTopLevel`.



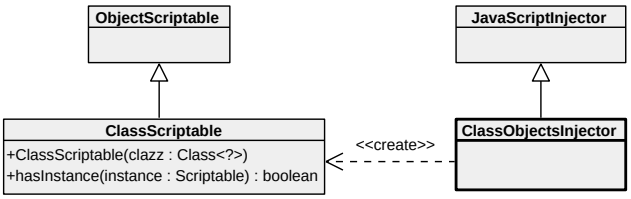
Obrázek 5.28: Diagram tříd objektu globálního scope

Pro implementaci objektu do scope využívá třída `ObjectTopLevel` instance implementátoru `ObjectImplementor`. Implementátor přijímá objekt, jenž má být implementován a členy třídy implementovaného objektu získané ze skriptovacího enginu. Implementátor tak ve skutečnosti zná všechny členy objektu `ObjectMembers`, které by měly být v globálním scope implementovány. Členy objektu jsou implementovány s využitím veřejných statických metod třídy `ObjectScriptable`: `defineObjectField()` a `defineObjectFunction()`. Třída `ObjectScriptable` je nádstavbou nad `ScriptableObject`, která umožňuje v rámci své instance uchovat nativní objekt Javy, jenž sama tímto zapouzdřuje. Při zavolání metody `defineObjectField()` vytváří třída objekt `HostedJavaField`, který následně do předaného scope vkládá. Po zavolání metody `defineObjectFunction()` vytváří třída nový objekt `HostedJavaMethod`, který taktéž vkládá do předaného scope. Nachází-li se v předaném scope již jiný existující objekt `HostedJavaMethod`, pak je metoda `ObjectFunction` přidána do listu přetížených metod existujícího objektu `HostedJavaMethod`.

Globální scope implementuje standardní objekty Javy a odstraňuje veškeré rozšiřující objekty Rhina, které by mohly narušit bezpečnost provádění skriptu, např. `JavaImporter`, `Packages`, `importClass`, `importPackage`, atd.

Z důvodu umožnění správné funkce operátoru `instanceof`, nebylo nutno do globálního scope implementovat pouze globální objekt `Window`, ale také všechny objekty tříd. Přidání této podpory zajistil injektor `ClassObjectInjector` (obrázek 5.29), který všechny třídy, jež by měly být viditelné ve skriptu, převádí na objekty `ClassScriptable`. Objekty

`ClassScriptable` zapouzdřují objekt třídy a předefinovávají metodu `hasInstance()`, která se volá při kontrole, zda je předaná instance typu, jenž zapouzdřuje objekt `ClassScriptable`.

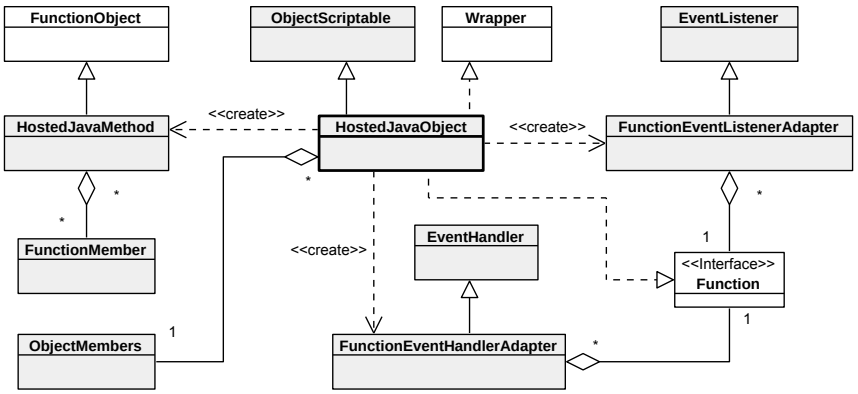


Obrázek 5.29: Diagram tříd injektoru globálního scope

5.3.3 Zapouzdření nativních objektů Javy

Implementováním globálního scope jsme získali množinu atributů a metod, které mohou navracet nativní objekty Javy uživatelskému skriptu. Skriptovací engine ovšem nativním objektům Javy „nerozumí“ a nedokáže s nimi pracovat, protože neimplementují rozhraní `Scriptable`. Knihovna Rhino řeší zapouzdřování objektů pomocí výchozí obalovací továrny `WrapFactory`, jejíž instance je uchovávána v objektu kontextu Rhina. Implementace obalovací továrny zapouzdřuje nativní objekty do objektu `NativeJavaObject`.

Jelikož nebylo možné ovlivnit exportovatelné členy nativních objektů, bylo nutné vytvořit novou obalující továrnu, která vytváří nový obal pro nativní objekty reprezentovaný třídou `HostedJavaObject` (obrázek 5.30). Obal nativního objektu `HostedJavaObject` zapouzdřuje nativní objekt, k němuž lze přistoupit např. rozbalením díky rozhraní `Wrapper`. Na rozdíl od globálního kontextu, obal neimplementuje všechny exportovatelné členy do scope, nýbrž vytváří pouze rozhraní pro přímý přístup do zapouzdřených nativních objektů. Obal obsahuje referenci na všechny exportovatelné členy `ObjectMemers` zapouzdřeného objektu a této reference využívá při potřebě získat nebo nastavit některou vlastnost zapouzdřeného objektu. Vlastnostmi zde máme na mysli i funkce zapouzdřeného objektu `ObjectFunction`, které předtím než je vrátíme z rozhraní `Scriptable`, musí být obaleny do nového objektu `HostedJavaMethod`. Kromě zajištění rozhraní `Scriptable` rozšířením třídy `ObjectScriptable`, obal `HostedJavaObject` implementuje také rozhraní `Function`. Rozhraní `Function` je implementováno za účelem umožnění volání exportovatelných konstruktorů zapouzdřeného objektu. Při volání operátoru `new` dochází k zavolání metody `construct()`, která je obsažena právě v rozhraní `Function`.

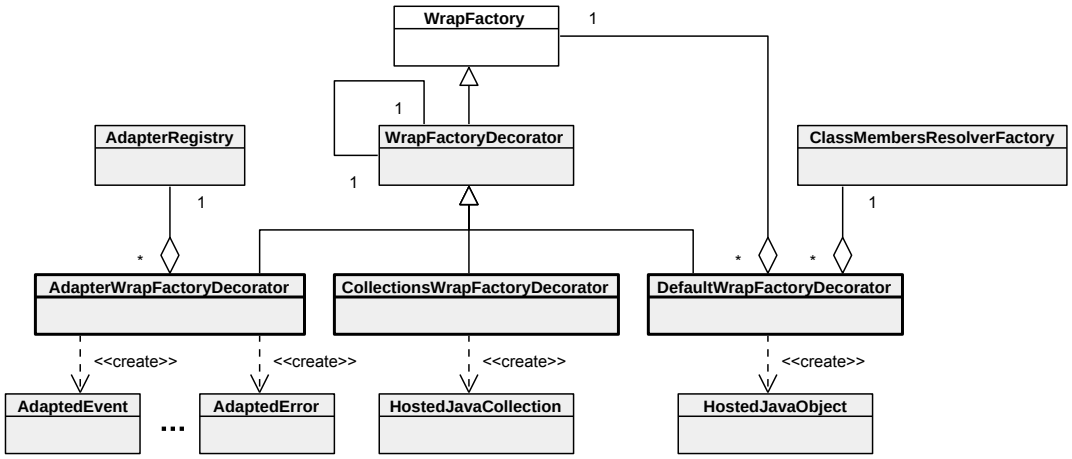


Obrázek 5.30: Diagram tříd zapouzdřující třídy nativních objektů Javy

Obalující objekt `HostedJavaMethod` rozšiřuje funkční objekt `FunctionObject` a umožňuje uchovávat všechny přetížené metody, jež tento objekt představuje. Z důvodu volání přetížených metod, byla předefinována metoda `call()`, která se provádí při volání funkce. Při invokaci funkce dochází nejprve k vyhledání metody v listu přetížených metod a pak k zavolání vybrané metody. Bylo implementováno jednoduché vyhledávání metod metodou `getNearestObjectFunction()`, která vyhledává funkce na základě počtu argumentů a jejich typů. Vyhledávání počítá i s možností volání metod s proměnnými argumenty.

Při nastavování nativních objektů JavaScriptu do některých rozhraní jádra uživatelského agenta, se naskytla potřeba některé tyto nativní objekty obalovat. V diagramu na obrázku 5.30 vidíme, že např. obal `HostedJavaObject` může vytvářet adaptující objekty `FunctionEventHandlerAdapter` a `FunctionEventListenerAdapter`. Tyto adaptéry jsou vytvářeny při požadavku na uložení nativní funkce JavaScriptu do některého z rozhraní `EventHandler` nebo `EventListener`. V aktuální implementaci není momentálně jednotné místo, které by se staralo o správu těchto adaptérů.

Výše popsaný obal nativních objektů Javy `HostedJavaObject` je vytvářen v rámci upravené obalující továrny `DafaultWrapFactoryDecorator` (obrázek 5.31). Vytvořená obalující továrna nově rozšiřuje třídu `WrapFactoryDecorator`, která umožňuje řetězení jednotlivých obalujících továren.



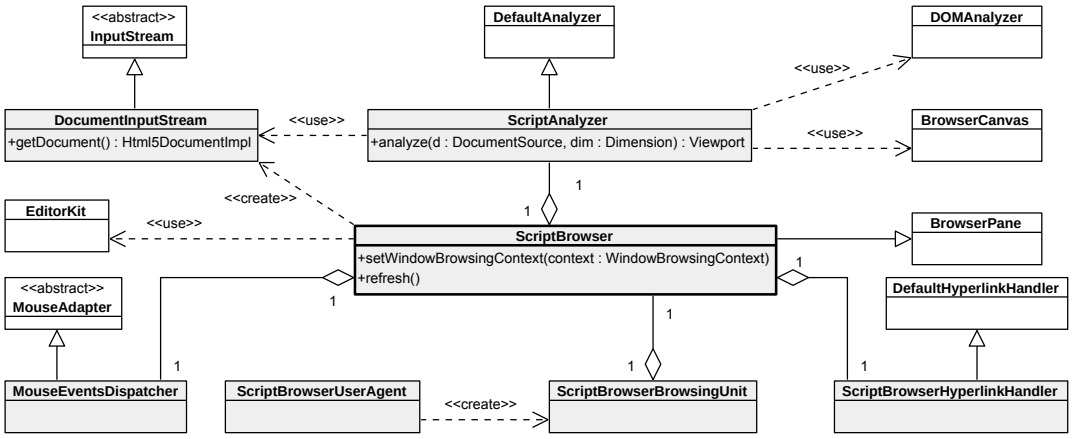
Obrázek 5.31: Diagram tříd obalující továrny

Pro JavaScriptový engine byly implementovány celkem tři obalující továrny, které se provádí v následujícím pořadí:

1. `AdapterWrapFactoryDecorator` – obalující továrna převádějící nativní objekty, jež mají korespondující adaptér v registru adaptéru, na adaptované objekty. Továrna vždy volá následující zřetězenou továrnu;
2. `CollectionsWrapFactoryDecorator` – obalující továrna převádějící nativní kolekce Javy na objekt HostedJavaCollection, čímž zpřístupňuje jejich prvky ve skriptech. Pokud není nativní objekt kolekcí, pak dochází k zavolání poslední zřetězené továrny;
3. `DafaultWrapFactoryDecorator` – obalující továrna provádějící převod všech nativních objektů na objekty `HostedJavaObject` tak, jak bylo popsáno výše.

5.4 Uživatelské rozhraní pro prohlížení stránek

V kapitole 5.1.4 jsme se zabývali zpracováním obsahu zdroje a jeho převodem na dokument. Dalším krokem v reálném prohlížeči by bylo zobrazení dokumentu uživateli a to již během jeho parsování. Jelikož není hotova podpora pro inkrementální a dynamické změny obsahu dokumentu, bylo implementováno zobrazování dokumentu až poté, co parser dokumentu dokončí svůj běh. Jednoduchou komponentu, která umožňuje prohlížení HTML dokumentů, implementuje třída `ScriptBrowser`. Komponenta vychází z komponenty `BrowserPane` implementované v projektu SwingBox.



Obrázek 5.32: Diagram tříd komponenty k vykreslování dokumentů

Třída `ScriptBrowser` zapouzdřuje procházecí kontext okna, nad kterým registruje odběratele pro události generované navigačním kontrolerem a společnou historií sezení. Přijme-li odběratel událost, která znamená změnu dokumentu, pak tento odběratel volá metodu `refresh()`, jež způsobí vykreslení dokumentu. Během vykreslení dokumentu je získána reference na objekt `EditorKit`, nad nímž je zavoláno čtení dokumentu metodou `read()`. Jelikož dokument je již zpracován, tak metodě `read()` tento dokument předáme zapouzdřený do vstupního datového toku `DocumentInputStream`. Metoda `read()` vnitřně pro vykreslení dokumentu využívá instance `ScriptAnalyzer`, která byla s komponentou `ScriptBrowser` asociována během její konstrukce. V rámci metody `analyze()` třídy ScriptAnalyzer dochází k získání zapouzdřeného dokumentu ze vstupního datového toku, analyzování objektu DOM třídou `DOMAnalyzer` a návrhu výsledného vzhledu dokumentu třídou `BrowserCanvas`.

Každá prohlížecí komponenta `ScriptBrowser` má asociovanou vlastní obslužnou rutinu pro to, co se má stát, když se klikne na URL odkaz. Obsluhu kliku na hyperlink implementuje třída `ScriptBrowserHyperlinkHandler`, která vnitřně volá navigační kontroler a statickou metodu `followHyperlink()`. Kromě obsluhy URL odkazů, komponenta implementuje generování událostí myši pro příslušné uzly dokumentu, nad kterými událost nastala. Jelikož komponenta může obsahovat posuvníky, informuje procházecí jednotka `ScriptBrowserBrowsingUnit` pomocí rozhraní `BarProp`, zda jsou posuvníky aktivni či nikoliv.

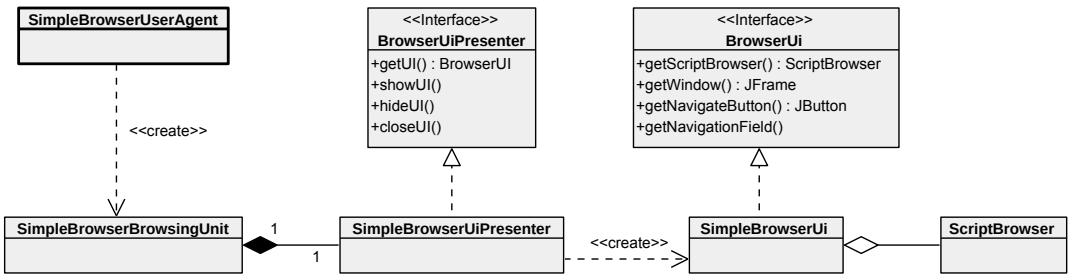
5.5 Ukázkové aplikace

Pro znázornění funkčnosti uživatelského agenta byly implementovány dvě ukázkové spustitelné aplikace, které demonstrují jeho základní funkce. Obě aplikace sloužily k otestování možnosti prohlížet HTML dokumenty. Jejich implementace se nachází v balíku projektu nazvaném `org.fit.cssbox.scriptbox.demo`.

Jednoduchá aplikace, která umožňuje pouze navigování stránek a procházení společné historie sezení, je podrobněji popsána v kapitole 5.5.1. Další implementovanou aplikací, která přidává některé ladící komponenty klientského JavaScriptu, se zabývá kapitola 5.5.2.

5.5.1 Jednoduchý prohlížeč

Pro vytvoření jednoduchého prohlížeče bylo zapotřebí definovat nového uživatelského agenta `SimpleBrowserUserAgent` (obrázek 5.33), který vytváří jednoduché procházecí jednotky `SimpleBrowserBrowsingUnit` obsahující uživatelské rozhraní. K implementaci jednoduchého uživatelského rozhraní bylo využito architektury MVP (model-view-presenter) s pasivním pohledem. Presenter v implementaci představuje třída `SimpleBrowserUIPresenter`, pohled třída `SimpleBrowserUI` a model uživatelský agent `SimpleBrowserUserAgent`.



Obrázek 5.33: Diagram tříd rozšíření uživatelského agenta o uživatelské rozhraní

Během konstrukce procházecí jednotky dochází nejprve k tvorbě presenteru, jenž v rámci své tvorby vytváří pohled, se kterým je přímo spjat. Jakmile je pohled vytvořen, dochází presenterem k zaregistrování odběratelů událostí nad uživatelskými komponentami pohledu. Kromě odběratelů uživatelských komponent registruje presenter též odběratele u objektů uživatelského agenta, tzn. modelu. Konkrétně jsou zaregistrovány odběratelé nad navigačním kontrolerem hlavního procházecího kontextu a nad společnou historií sezení procházecí jednotky. Po zachycení událostí z navigačního kontroleru nebo společné historie sezení dochází presenterem k pasivní úpravě pohledu. Při dokončení stažení dokumentu je např. aktualizován název aplikace podle titulku stránky, po průchodu společnou historií sezení jsou aktualizovány tlačítka vpřed a zpět.

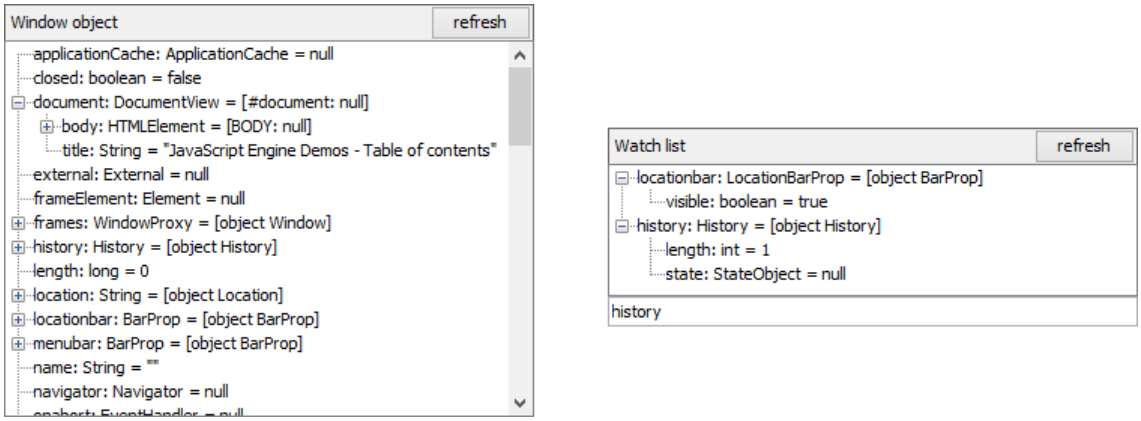
Výsledný jednoduchý prohlížeč je zobrazen na obrázku v příloze E. Prohlížeč umožňuje navigovat nové stránky pomocí navigačního řádku a procházet společnou historii sezení pomocí tlačítek vpřed a zpět.

5.5.2 Tester JavaScriptu

Během testování funkčnosti uživatelského agenta se naskytla potřeba pro ladění klientských skriptů JavaScriptu – vznik tzv. testeru. Z tohoto důvodu bylo implementováno rozšíření jednoduchého prohlížeče uvedeného v předešlé kapitole 5.5.1. Rozšíření přidalo ke komponentě pro prohlížení dokumentů další tři nové komponenty. Architektura pro tester byla

zvolena také typu MVP. Presenter `JavaScriptTesterUIPresenter` využívá implementované funkčnosti v presenteru `SimpleBrowserUIPresenter`. Vzhled testeru pohledu byl vytvořen zcela nový – třída `JavaScriptTesterUI`. Vzhled implementuje rozhraní `BrowserUI` společné pro pohled jednoduchého prohlížeče.

První dvě komponenty, které byly navrženy a zcela implementovány umožňují procházet objekty, které jsou definované ve skriptu. Na obrázku 5.34 můžeme vidět komponentu `ScriptObjectViewer` a komponentu `ScriptObjectsWatchList`.

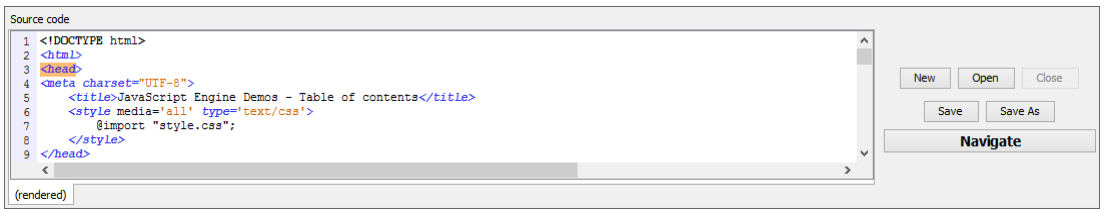


Obrázek 5.34: Komponenta `ScriptObjectViewer` (zleva) a `ScriptObjectsWatchList`

První komponenta `ScriptObjectViewer` umožňuje zobrazit všechny atributy objektu, který se jí předá. Společným objektem pro všechny klientské skripty je objekt `Window`, jehož bylo právě zvoleno pro výpis v aplikaci testeru. Jednotlivé položky ve výpisu nelze ovlivnit a jinak modifikovat. Komponenta je pouze informativní a ukazuje aktuální stav objektu `Window`.

Druhá komponenta `ScriptObjectsWatchList` přidává možnost volby objektů, které mají být ve výpisu objektů zobrazeny. Objekty lze kdykoliv přidávat nebo je odebírat.

Třetí komponenta (obrázek 5.35) slouží k procházení staženého zdrojového kódu dokumentu a k jeho případné modifikaci. Pro zvýraznění syntaxe byla použita získaná knihovna `jsyntaxpane`. V rámci komponenty lze dále vytvářet záložky, do kterých můžeme vkládat zdrojový kód, jenž je možno uložit nebo navigovat. Navigace vytvořeného zdrojového kódu probíhá tak, že se zdrojový kód uloží do složky dočasných souborů, ze které pak probíhá samotná navigace.



Obrázek 5.35: Komponenta pro zobrazování zdrojového kódu dokumentu

Výsledná podoba aplikace je znázorněna na obrázku v příloze F.

Kapitola 6

Testování a dosažené výsledky

Následující kapitola se věnuje testování implementovaného řešení rozšíření projektu SwingBox a hodnotí jeho výkonost či kompatibilitu. První podkapitola 6.1 se zaměřuje na způsob, jakým testování probíhalo, a uvádí testované aspekty řešení. Další podkapitola 6.2 hodnotí výsledné řešení z hlediska výkonnosti. Míry ukazatele výkonnosti jsou porovnány s nejznámějšími internetovými prohlížeči. V závěru kapitoly jsou shrnuty dosažené výsledky, zhodnocena kompatibilita řešení a uvedeny některé další možnosti, jakými by mohl v budoucnu pokračovat další vývoj celého projektu.

6.1 Testování řešení

Testování implementovaného řešení probíhalo více způsoby. Funkčnost byla testována během celého vývoje rozšíření. Některé testy byly automatizovány a nyní se nachází v balíku `tests`. Automatizované testy byly vytvořeny pro otestování událostní smyčky, jednotlivých resolverů členů tříd a zapouzdřujících objektů nativních objektů Javy. Dodatečně byla funkčnost ověřena pomocí benchmarku SunSpider a ukázkovými HTML stránkami vytvořenými pro ukázkové aplikace zahrné v rozšíření.

V jednoduchých testech funkce událostní smyčky `EventLoopTests` byly vytvořeny tři testovací případy, které ověřují nejdůležitější funkčnost událostní smyčky. První případ `TestSpinEventLoopOrder` testuje, zda dochází ke správnému přerušení právě probíhající úlohy, předpokládané rotaci událostní smyčky a opětovnému vložení úlohy do fronty úloh. Druhý případ `TestSpinEventLoopAbort` testuje možnost ukončení událostní fronty metodou `abort()`, což hraje důležitou roli při rušení procházecí jednotky. Poslední případ `TestRoundRobinScheduler` ověřuje správnou funkci plánovače úloh s kruhovým výběrem a testuje, zda dochází opravdu ke kruhovému vybírání úloh.

Další implementované testy `HostedJavaObjectTest` a `JavaScriptAnnotationTests` sloužily k otestování funkce implementátoru globálního scope a exportu nativních objektů Javy. Byl otestován výchozí resolver členů tříd a resolver založený na skriptovacích anotacích. Testy byly cíleny k ověření správného vyjmenování výčtových vlastností, tzn. členů, jež měly nastavenou možnost `ENUMERABLE`. Dále se testovaly možnosti `FIELD_GET_OVERRIDE`, `FIELD_SET_OVERRIDE`, `CALLABLE_GETTER` a `CALLABLE_SETTER`. Testy u všech členů tříd zkontrolovaly jejich návratové hodnoty s předpokládanými hodnotami a ověřily, zda byl exportován správný počet vlastností. Všechny vlastnosti, které by neměly být vidět ve skriptech, byly otestovány na nedefinovanou hodnotu `undefined`.

Poslední implementované testy `CollectionsWrapFactoryTests` ověřily správnou funkci konverze nativních kolekcí Javy na objekty, které umožňují přístup k prvkům kolekci pomocí indexů.

Jak již bylo zmíněno, k otestování správné funkčnosti sloužil i benchmark SunSpider[1], který ověřil správnou implementaci JavaScriptového enginu podle normy JSR 223. Benchmark obsahuje řadu testů k ověření základní JavaScriptové množiny a vyhodnocení výkonnosti skriptovacího enginu (kapitola 6.2).

Důležitou roli během testování představovaly ukázkové HTML stránky, které slouží pro demonstraci funkčnosti implementovaného rozhraní klientského JavaScriptu. Během tvorby ukázkových stránek se otestovaly všechny rozhraní, které tyto stránky využívaly.

Bylo zhotoveno celkem 6 ukázek znázorňujících následující stěžejní implementace:

1. `dialogs.html` – představení základních uživatelských dialogů,
2. `events.html` – ukázka generování některých DOM událostí,
3. `scripts.html` – spouštění skriptů v rámci dokumentu,
4. `history.html` – procházení historií sezení,
5. `location.html` – navigovaní dokumentů,
6. `sunspider.html` – vnoření dokumentu pomocí značky `<iframe>`.

6.2 Vyhodnocení výkonnosti

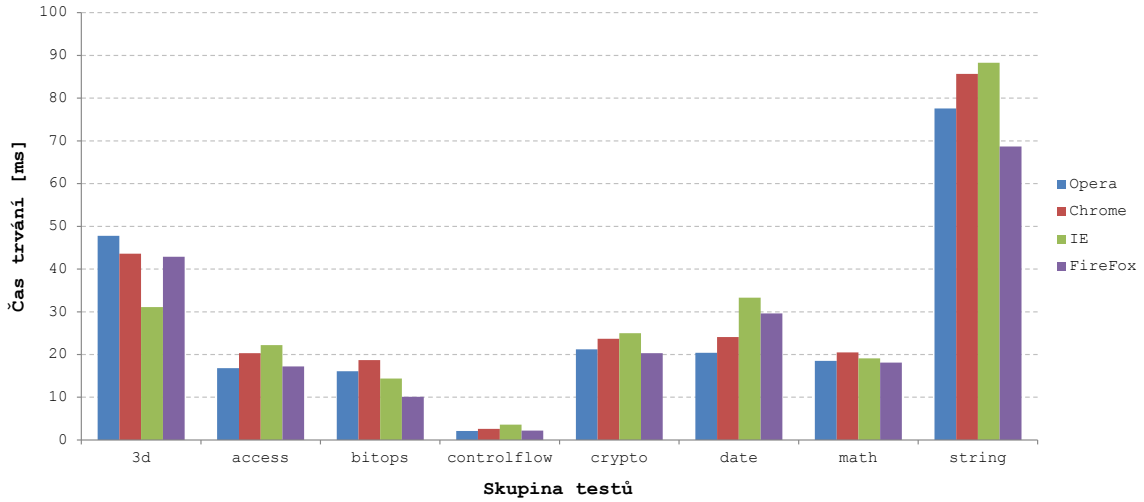
Po dokončení implementace a jejím otestování byla vyhodnocena výkonnost implementovaného řešení. Pro všechny testy, které sloužily k vyhodnocení výkonnosti, bylo využito nezatíženého stroje, jehož výpočetní zdroje nebyly limitovány, např. nedostatkem operační paměti. Testovaná aplikace s implementovaných rozšíření byla přeložena překladačem EJC[2]. Pro testování sloužil stroj s operačním systémem Windows 8 a s následovnou základní konfigurací: Intel Core i5 2430M Sandy Bridge, RAM 4GB DDR3.

Pro vyhodnocení výkonnosti bylo využito benchmarku SunSpider, který obsahuje sadu základních výkonnostních testů. Jednotlivé testy jsou orientovány do řady vědních oborů a problémů reálného světa. Benchmark obsahuje například testy vykonávající kryptografické operace (SHA1, MD5, AES), sledování paprsku v prostoru (angl. 3D ray tracing), dekompresi spustitelného kódu aj.

Abychom mohli získané výsledky benchmarku porovnat s námi naměřenými hodnotami, bylo provedeno nejprve měření pro nejpoužívanější internetové prohlížeče. Vyhodnocenými prohlížeči jsou: Firefox, Chrome, Internet Explorer (dále IE) a Opera. Pro získání naměřených hodnot bylo využito veřejně přístupného API pro spuštění jednotlivých testů benchmarku. Každý test byl spuštěn celkem 5 krát. Získané časy trvání pro jednotlivé skupiny testů a pro všechny zmíněné prohlížeče lze vidět na obrázku 6.1.

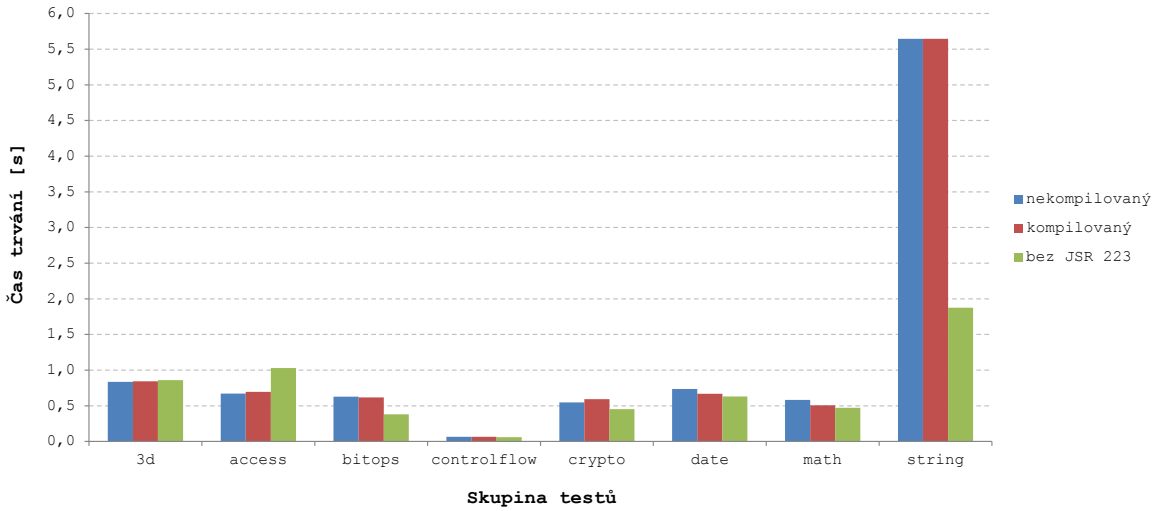
Z grafu 6.1 bylo zjištěno, že výkonnosti jednotlivých skriptovacích enginů nejpoužívanějších internetových prohlížečů se výrazně neliší. Testy nejlépe dopadly pro prohlížeč FireFox, který všechny testy dokončil průměrně za 209 milisekund, a nejhůře pro prohlížeč IE, jehož testy trvaly průměrně 240 milisekund.

[1] https://www.webkit.org/perf/sunspider/sunspider.html
[2] http://www.eclipse.org/jdt/core/



Obrázek 6.1: Trvání jednotlivých skupin testů v prohlížečích

Pro otestování námi implementovaného řešení bylo nutno vytvořit vlastní API benchmarku, které spouštělo jednotlivé výkonnostní testy. Veřejné API nebylo možné použít, jelikož pro svůj běh využívalo podmnožinu JavaScriptu, která nebyla doposud implementována. Vytvořené vykonávací API se nachází ve složce `benchmark` umístněné ve složce pro zdrojové kódy ukázkových HTML stránek (příloha A). Implementované API získává ze zadané URL adresy dokument, který obsahuje seznam všech testů, jež by se měly vykonat. Skripty, jejichž cesty jsou uvedené v daném listu staženého dokumentu, se postupně jednotlivě vykonávají s předem známým počtem opakování.



Obrázek 6.2: Trvání jednotlivých skupin testů v implementovaném řešení

Implementované řešení jsme nejprve sestavili tak, aby vykonávané skripty nebyly kompilovány před jejich spuštěním. Podruhé jsme řešení sestavili tak, že všechny skripty byly kompilovány. Nad oběma sestaveními jsme nechali běžet benchmark. Dále bylo provedeno několik experimentů, jelikož poslední skupina testů dopadla výrazně pod očekávání. Výsledkem experimentů bylo odstranění obalu `ScriptContextScriptable`, který zajišťoval

data binding pro standardního skriptovacího API. Důsledkem odstranění obalu bylo např. znemožnění injekce skriptovacího kontextu. Tento zásah nepředstavoval žádný problém pro běh benchmarku. Nakonec jsme provedli i benchmark nad skriptovacím enginem, který neměl implementované rozhraní podle normy JSR 223. Výsledky všech tří běhů benchmarku jsou znázorněny na obrázku 6.2. Každý test benchmarku byl spuštěn celkem 15 krát a jeho doba trvání byla zprůměrována.

Získané časy trvání byly oproti ostatním internetovým prohlížečům výrazně delší. Kompilace skriptu neměla zásadní vliv na zrychlení běhu skriptu. Z důvodu „váznoucí“ poslední skupiny testů `string` byl celkový čas běhu jednoho benchmarku s kompilovaným nebo nekompilovaným kódem 9, 6 sekundy. Implementovaný skriptovací engine byl zhruba 42 krát pomalejší než skriptovací enginy ostatních měřených internetových prohlížečů. Odstraněním standardního skriptovacího API došlo k výraznému zlepšení poslední skupiny testů `string`. V ostatních testovaných skupinách došlo taktéž k mírnému zlepšení. Celkový čas byl 5, 7 sekundy, což je 25 krát pomalejší než průměrný celkový čas u dalších internetových prohlížečů. Problém vzniklý v poslední skupině `string` nebyl blíže trasován. Vysoká časová latence je s největší pravděpodobností způsobena převodem optimalizovaných textových řetězců JavaScriptu `ConsString` na nativní textové řetězece Javy `String`, který probíhá při každém uložení řetězce do globálního scope.

6.3 Zhodnocení řešení a jeho kompatibility

Implementace navrženého řešení proběhla podle specifikace HTML 5.1, jež je stále ve fázi editorského návrhu. Stále rozšiřovaný a opravovaný dokument návrhu specifikace HTML 5.1 byl konfrontován se starší verzí specifikace HTML 5, která je již ve fázi doporučení. Až na některé výjimky lze říci, že dokončená podmnožina podpory klientského JavaScriptu je kompatibilní se specifikací HTML 5.1. Všechny vzniklé nekompatibility byly vyznačeny v kódu odpovídajícím komentářem. Nekompatibility vznikly hlavně z důvodu nereentrantního parseru nebo jiné chybějící implementace. Mezi některé nejpodstatnější nekompatibility můžeme považovat např.:

1. **neblokující spouštění skriptů** – nebylo implementováno čekání na stažení kaskádových stylů dokumentu, které by mělo blokovat vykonávání skriptů. Stahování těchto zdrojů zajišťuje jiná třída v rámci projektu CSSBox. Abychom zabránili vícenásobnému stahování zdrojů, je nutné v budoucnu rozhraní projektů CSSBox a ScriptBox sjednotit. Zbylé požadavky na pořadí spouštění skriptů byly již implementovány přesně podle uvedené specifikace;

2. **nemožnost reentrantního parsování** – aktuálně využívaný parser dokumentů NekoHTML není reentrantní. Chybějící funkčnost velice komplikuje případnou budoucí implementaci např. volání `document.write()`. Některé mechanismy, které ošetřují např. vnořené vložení `<script>` značky, nebyly kompletně implementovány, protože parser není reentrantní.

V porovnání s předešlým projektem SwingBox, v němž chyběla implementace jádra uživatelského agenta, se snaží aktuální projekt věrně následovat specifikaci. Následně uvedeme některé případy, které vylepšují implementaci projektu SwingBox.

V projektu SwingBox definovaná obslužná třída `DefaultHyperlinkHandler` pro klik na URL odkaz byla předefinována třídou `ScriptBrowserHyperlinkHandler`. Nová implementace obsluhy kliku na URL odkaz využívá jádra asociovaného uživatelského agenta, tzn.

navigačního kontroleru pro navigování dokumentů. Díky změně implementace lze například odkazovat pomocné procházecí kontexty.

Projekt SwingBox implementoval stahování dokumentu v rámci událostní smyčky uživatelského rozhraní, což mělo za následek zablokování dané událostní smyčky a znemožnění práce s uživatelským rozhraním. Aktuální implementaci stahování dokumentu zajišťuje třída navigačního kontroleru, která naviguje dokument v asynchronním vlákně a pro jeho zpracování vkládá úlohu do událostní fronty aktuální procházecí jednotky. Celý proces stažení dokumentu a jeho zpracování probíhá v jiných vláknech než je vlákno uživatelského prostředí. Uživatelské rozhraní je nyní plně responsibilní.

Implementace jádra uživatelského agenta probíhala podle jednotlivých podkapitol specifikace [7]. Jmenovitě bylo čerpáno z následujících částí specifikace:

1. **Webové aplikační API** [23] – využívané pro implementaci základních konceptů pro skripty `Script` a nastavení skriptů `ScriptSettings`. Část specifikace inspirovala také pro definici událostní smyčky `EventLoop`, událostí `Task` a jednoduchých uživatelských dialogů;
2. **Načítání webových stránek** [10] – sloužilo pro implementaci procházecích jednotek, procházecích kontextů, navigování dokumentů a procházení historie sezení. Podkapitola specifikace též definovala rozhraní `Window`, `Location` a `History`;
3. **Skriptování** [19] – definovalo způsob zpracovávání elementů `<script>`, jejich převod na skripty `Script` a pořadí spouštění skriptů;
4. **HTML syntaxe** [8] – sloužilo pro změnu NekoHTML parseru tak, aby správně zajišťoval spouštění parsovacích skriptů HTML dokumentu;
5. **Společná infrastruktura** [2] – inspirace pro způsob stahování zdrojů.

Závěrem uvedeme kompletní implementovanou podmnožinu klientského JavaScriptu. Byla implementována následující podmnožina:

1. **globální objekt** – částečná implementace rozhraní `Window` (kapitola 5.1.8),
2. **navigování dokumentů** – implementace rozhraní `Location` (kapitola 5.1.5),
3. **procházení historie** – implementace rozhraní `History` (kapitola 5.1.6),
4. **manipulace s URL** – z větší části hotová implementace rozhraní `URLUtils` a `URL` (kapitola 5.1.10),
5. **DOM události** – jsou generovány některé události jádra uživatelského agenta, ale i události způsobené uživatelem, např. klikem na element dokumentu (kapitola 5.1.9).

6.4 Možnosti budoucího vývoje

Implementované řešení lze dále rozšiřovat. Možných cest dalšího vývoje vytvořeného rozšíření projektu SwingBox je celá řada. Následující vývoj by měl být cílen zejména na přidání další množiny funkčnosti klientského JavaScriptu. V některých částech již existující implementace je prostor pro různé optimalizace nebo pro dokončení a vylepšení funkčnosti. Malou část jednotlivých možností dalšího vývoje shrneme v následujícím seznamu. V některých případech naznačíme i možné řešení daného problému.

Budoucí vývoj by se mohl zaměřit například na následující možnosti:

1. **další podmnožina JavaScriptu** – mělo by být implementováno např. rozhraní pro časovače, rozhraní `Navigator`, `ApplicationCache`, `External` aj.;
2. **rozhraní objektu dokumentu** – objekt `Window` v současné době obsahuje odkaz na dokument, který není zcela viditelný klientskému JavaScriptu. Rozhraní dokumentu implementuje pouze nejpodstatnější funkce, které bylo zapotřebí implementovat za účelem otestování některých částí uživatelského agenta. V budoucnu se předpokládá s odstraněním těchto metod z implementace dokumentu, kterou by mělo vidět pouze jádro uživatelského agenta, a vytvoření adaptéru nad tímto dokumentem. Adaptér by měl zpřístupnit viditelné rozhraní pro dokument v klientském kódu JavaScriptu. Vývojem viditelného rozhraní pro dokument se zabývala paralelně běžící diplomová práce Bc. Radima Kocmana. Pro propojení by stačilo vytvořit odpovídající adaptér a zaregistrovat ho do registru adaptérů `AdapterRegistry`. Do viditelného rozhraní by musely být přidány skriptovací anotace;
3. **vylepšení** `WindowProxy` – aktuální rozhraní `WindowProxy` slouží pro přístup ke globálnímu objektu `Window` aktivního dokumentu aktuálního procházecího kontextu. V implementaci rozhraní `WindowProxy` zpřístupňuje chování objektu `Window`. Podle specifikace by objekt `WindowProxy` měl mít chování zcela totožné jako objekt `Window`. V aktuální implementaci požadavku není zcela vyhověno, jelikož objekt `Window` a objekt `WindowProxy` jsou dvě odlišné instance. Implementačně proto selhává například operátor identity, neplatí `window === this`. Dalším problémem je, že definice nové proměnné uvnitř objektu `WindowProxy` se neprovádí v globálním scope, ale ve scope objektu `WindowProxy`. Problém by se dal vyřešit například předefinováním metody `shallowEq()` ve třídě `ScriptRuntime`. Tuto metodu vnitřně volá třída interpretu Rhina. Problémem je, že modifikace těchto tříd není veřejně přístupná. Přesměrování scope objektu `WindowProxy` na globální scope aktivního objektu `Window` lze řešit přepsáním metod `get()` a `set()` rozhraní `Scriptable` pro objekt `WindowProxy`;
4. **mezipaměť pro členy tříd** – při každém přístupu k nativnímu objektu Javy nyní dochází k vyhodnocování členů, jež by měly být exportovány do klientského JavaScriptu. Členy tříd pro již vyhodnocené třídy by bylo výhodné uchovávat v mezipaměti. Pro příští zapouzdření stejného nativního objektu Javy by se mělo využívat právě členů tříd uchovaných v této mezipaměti;
5. **podpora** `[Replaceable]` – k WebIDL atributu `[Replaceable]` v implementaci koresponduje zjednodušená možnost členů tříd `PERNAMENT`, kterou nyní resolver anotovaných členů tříd neumožňuje nastavovat a vynechávat. Možnost `PERNAMENT` je proto v aktuální implementaci vždy nastavena. S nenastavenou možností `PERNAMENT` nepředpokládá ani implementace pro obalování nativních objektu Javy do kódu JavaScriptu. Bylo by nutné udělat také podporu pro přepisování exportovaných členů tříd nativních objektů Javy;
6. **další události DOM** – implementovat rozhraní pro nové DOM události, tak je definuje specifikace a ve správný okamžik je generovat. Chybí implementace například pro rozhraní `PageTransitionEvent` nebo `BeforeUnloadEvent`;
7. uvolňování dokumentu, ukončování právě spuštěných skriptů, podpora pro navigaci dalších URL schémat a jiných zdrojů než jsou HTML dokumenty.

Před implementací některých z výše zmíněných rozšíření bude nutno nejprve zvážit a vyhodnotit, zda nebude výhodnější přejít na nový skriptovací engine Nashorn, který během vývoje této práce oficiálně vyšel společně s JDK 1.8. Přechod na nový skriptovací engine by obnášel novou implementaci podpory pro tento skriptovací engine. Muselo by se znovu řešit zapouzdřování nativních objektů Javy, implementace globálního scope a další implementace nacházející se v balíku `org.fit.cssbox.scriptbox.script.javascript`.

Pokud by vývoj dále probíhal s enginem Nashorn, bylo by možné využít jeho podobné rozhraní pro všechny objekty JavaScriptu. Rhino pro všechny objekty JavaScriptu implementuje rozhraní `Scriptable`, Nashorn zavádí rozhraní `JSObject`. Jelikož jsou obě rozhraní sobě velmi podobná, lze tedy zvážit i vytvoření přemostění mezi knihovnou Rhino a Nashorn. Návrh přemostění je otázkou budoucího vývoje. Kompletní implementace nového podporovaného klientského skriptovacího enginu by mohla probíhat obdobně tak jako pro skriptovací engine využívající Rhino.

Kapitola 7

Závěr

Cílem této diplomové práce bylo se seznámit s projektem CSSBox a zanalyzovat problematiku skriptování v jazyce JavaScript z jazyku Javy. Na základě získaných teoretických znalostí navrhnout architekturu a způsob integrace skriptovacího stroje do projektu CSSBox. Dále tuto architekturu implementovat jako volitelné rozšiřitelní knihovny CSSBox.

V úvodní části textu bylo nejprve analyzováno skriptování v HTML dokumentech a referenční způsob pro přidání skriptovací podpory do dokumentu. Byl představen projekt CSSBox, do kterého proběhla integrace rozšíření, a komponenta SwingBox pro zobrazování HTML dokumentu.

V kapitole analýzy skriptovacích strojů byly demonstrovány existující řešení pro skriptování v jazyce JavaScript v Java aplikacích. Podrobně byla rozebrána technika tvorby skriptů s API knihovny Rhino. V kontrastu na tvorbu skriptů s nestandardním API Rhina, byla ukázána i tvorba skriptu se standardním skriptovacím API Javy.

Návrh integrace rozšíření skriptování v JavaScriptu přímo reflektoval teoretickou analýzu z předchozích kapitol. Z důvodu spouštění skriptů v dokumentu jsme museli navrhnout jeho postupné načítání. Abychom odstranili potřebu měnit kód skriptovacího stroje při přidávání nového rozšíření hlavního scope, navrhli jsme injektování do skriptovacího stroje s využitím injektáže závislostí. Jelikož knihovna Rhino obsahovala pouze základní kontrolu přístupu do Javy z JavaScriptu, navrhli jsme také třídy pro rozmělnění bezpečnosti a lepší správu bezpečnostních politik. Závěrem návrhu jsme demonstrovali příklad „registrace“ samotného rozšíření do projektu SwingBox.

Po abstraktním návrhu práce popisovala konkrétní implementované řešení. Bylo implementováno jednoduché jádro uživatelského agenta. Jádro umožnuje stahovat předem neznámé typy zdrojů. V závislosti na typu komunikačního protokolu dochází k požadované obsluze zdroje. Stažené zdroje jsou jádrem zpracovávány a převáděny na objekty (X)HTML dokumentů. O převod se stará obecný mechanismus, který je adaptivní v závislosti na typu obsahu, jenž byl ze zdroje stažen. V jádře byl implementován proces navigace dokumentů, který využívá předešle zmíněnou funkčnost na stažení obsahu dokumentu a jeho zpracování. Dokumenty lze jádrem procházet díky zabudované historii sezení, která je unikátní pro každý procházecí kontext. V jádře byl implementován základní mechanismus pro generování DOM událostí, přičemž některé události jsou již nyní jádrem podporovány. Jádro též zajišťuje spouštění skriptů. Byly implementovány všechny typy spouštění skriptů podle specifikace HTML 5. Skripty mohou být spuštěny v pořadí během parsování dokumentu, po dokončení parsování dokumentu nebo případně po načtení celého dokumentu. Implementována je podpora i pro asynchronní spouštění skriptů a dynamické vkládání skriptů do dokumentu.

Výše popsaná funkčnost jádra uživatelského agenta sloužila pro implementaci základních viditelných rozhraní JavaScriptu, jako jsou např.: `Location`, `History`, `Window`, `URL`, `URLUtils` aj. Rozhraní JavaScriptu jsou silně spjata s jádrem, zatímco jádro samotné s nimi minimálně. Rozhraní byla do klientského JavaScriptu exportována z nativních objektů Javy. Export nativních objektů zahrnoval prohledání tříd nativních objektů a rozhodnutí, zda má být člen tříd exportován či nikoliv. Prohledávání a vyhodnocení exportovatelných členů tříd bylo implementováno s využitím reflexe Javy a speciálních anotací, které značily exportovatelnost.

Implementovaný klientský JavaScriptový engine využívá navržené abstraktní architektury pro skriptovací enginy. Provádění skriptů zajišťuje knihovna Rhino, která byla rozšířena o standardní skriptovací API. Vytvořený klientský skriptovací engine umožňuje implementaci globálního objektu JavaScriptu do globálního scope a export nativních objektů Javy. Během implementace enginu byl brán ohled na bezpečnost. Z tohoto důvodu byly některé funkce knihovny Rhino zcela vynechány nebo předefinovány.

Závěrem diplomové práce byly zhodnoceny dosažené výsledky a provedeno testování implementovaného rozšíření. Testování proběhlo automatizovanými testy s využitím knihovny `JUnit`. Správná funkčnost implementovaného řešení byla ověřena i pomocí ukázkových HTML stránek. V rámci výkonnostních testů bylo zjištěno, že interpret Rhino je značně pomalejší než interprety nejpoužívanějších internetových prohlížečů. Poslední kapitola práce se věnovala možným budoucím rozšíření této práce.

Literatura

[1] *Bean Scripting Framework* [online]. [cit. 2013-12-15]. Dostupné z: <http://commons.apache.org/proper/commons-bsf/>.

[2] *Common infrastructure – HTML 5.1 Nightly* [online]. [cit. 2014-05-07]. Dostupné z: <http://www.w3.org/html/wg/drafts/html/master/infrastructure.html>.

[3] *CSSBox* [online]. [cit. 2014-01-10]. Dostupné z: <http://cssbox.sourceforge.net/>.

[4] *CyberNeko HTML Parser* [online]. [cit. 2014-01-10]. Dostupné z: <http://nekohtml.sourceforge.net/>.

[5] *Free EcmaScript Interpreter* [online]. [cit. 2013-12-10]. Dostupné z: <http://www.lugrin.ch/fesi/>.

[6] *Google Juice* [online]. [cit. 2014-01-10]. Dostupné z: <http://code.google.com/p/google-guice/>.

[7] *HTML 5.1 Nightly* [online]. [cit. 2014-05-07]. Dostupné z: <http://www.w3.org/html/wg/drafts/html/master/>.

[8] *The HTML syntax – HTML 5.1 Nightly* [online]. [cit. 2014-05-07]. Dostupné z: <http://www.w3.org/html/wg/drafts/html/master/syntax.html>.

[9] *jStyleParser* [online]. [cit. 2014-01-10]. Dostupné z: <http://cssbox.sourceforge.net/jstyleparser/>.

[10] *Loading Web pages – HTML 5.1 Nightly* [online]. [cit. 2014-05-07]. Dostupné z: <http://www.w3.org/html/wg/drafts/html/CR/browsers.html>.

[11] *New in Rhino 1.7R3* [online]. [cit. 2013-12-11]. Dostupné z: <https://developer.mozilla.org/en-US/docs/New_in_Rhino_1.7R3>.

[12] *New in Rhino 1.7R4* [online]. [cit. 2013-12-11]. Dostupné z: <https://developer.mozilla.org/en-US/docs/New_in_Rhino_1.7R4>.

[13] *Project Nashorn* [online]. [cit. 2013-12-10]. Dostupné z: <http://openjdk.java.net/projects/nashorn/>.

[14] *Rhino* [online]. [cit. 2013-12-10]. Dostupné z: <https://developer.mozilla.org/en-US/docs/Rhino>.

[15] *Rhino – na rozhraní JavaScriptu a Javy* [online]. [cit. 2014-01-10]. Dostupné z: <http://www.zdrojak.cz/clanky/rhino-na-rozhrani-javascriptu-a-javy/>.

[16] *Rhino History* [online]. [cit. 2013-12-10]. Dostupné z: <https://developer.mozilla.org/en-US/docs/Rhino/History>.

[17] *Rhino overview* [online]. [cit. 2013-12-11]. Dostupné z: <https://developer.mozilla.org/en-US/docs/Rhino/Overview>.

[18] *Rhino scopes and contexts* [online]. [cit. 2013-12-11]. Dostupné z: <https://developer.mozilla.org/en-US/docs/Rhino/Scopes_and_Contexts>.

[19] *Scripting – HTML 5.1 Nightly* [online]. [cit. 2014-01-11]. Dostupné z: <http://www.w3.org/html/wg/drafts/html/master/scripting-1.html>.

[20] *Standard ECMA-262* [online]. [cit. 2014-05-20]. Dostupné z: <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.

[21] *SwingBox* [online]. [cit. 2014-01-10]. Dostupné z: <http://cssbox.sourceforge.net/swingbox/>.

[22] *URL Standard* [online]. [cit. 2014-05-20]. Dostupné z: <http://url.spec.whatwg.org/>.

[23] *Web application APIs – HTML 5.1 Nightly* [online]. [cit. 2014-05-07]. Dostupné z: <http://www.w3.org/html/wg/drafts/html/master/webappapis.html>.

[24] *Web IDL* [online]. [cit. 2014-05-20]. Dostupné z: <http://www.w3.org/TR/WebIDL/>.

[25] *Yet Another Javascript Interpreter* [online]. [cit. 2013-12-10]. Dostupné z: <http://code.google.com/p/yaji-ecmascript-interpreter/>.

[26] BIELIK, P. *Rozhraní renderovacího stroje v jazyce Java* (diplomová práce). Brno: Vysoké učení technické, Fakulta informačních technologií, 2011. 50 s.

[27] BOSANAC, D. *Scripting in Java^{TM}*. TAUB, M. Boston: Pearson Education, Inc., 2008. ISBN 978-0-321-32193-0.

Přílohy

Seznam příloh

Příloha A

Obsah přiloženého DVD

Elektronický nosič obsahuje pracovní adresář pro vývojové prostředí Eclipse Kepler. Implementované rozšíření se nachází ve složce projektu **ScriptBox/**. Ostatní projekty umístěné v adresáři jsou zapotřebí ke správné kompilaci knihovny rozšíření. Manuál ke kompilaci knihovny lze nalézt buď v souboru **README**, v příloze C nebo na stránkách repozitáře projektu[1].

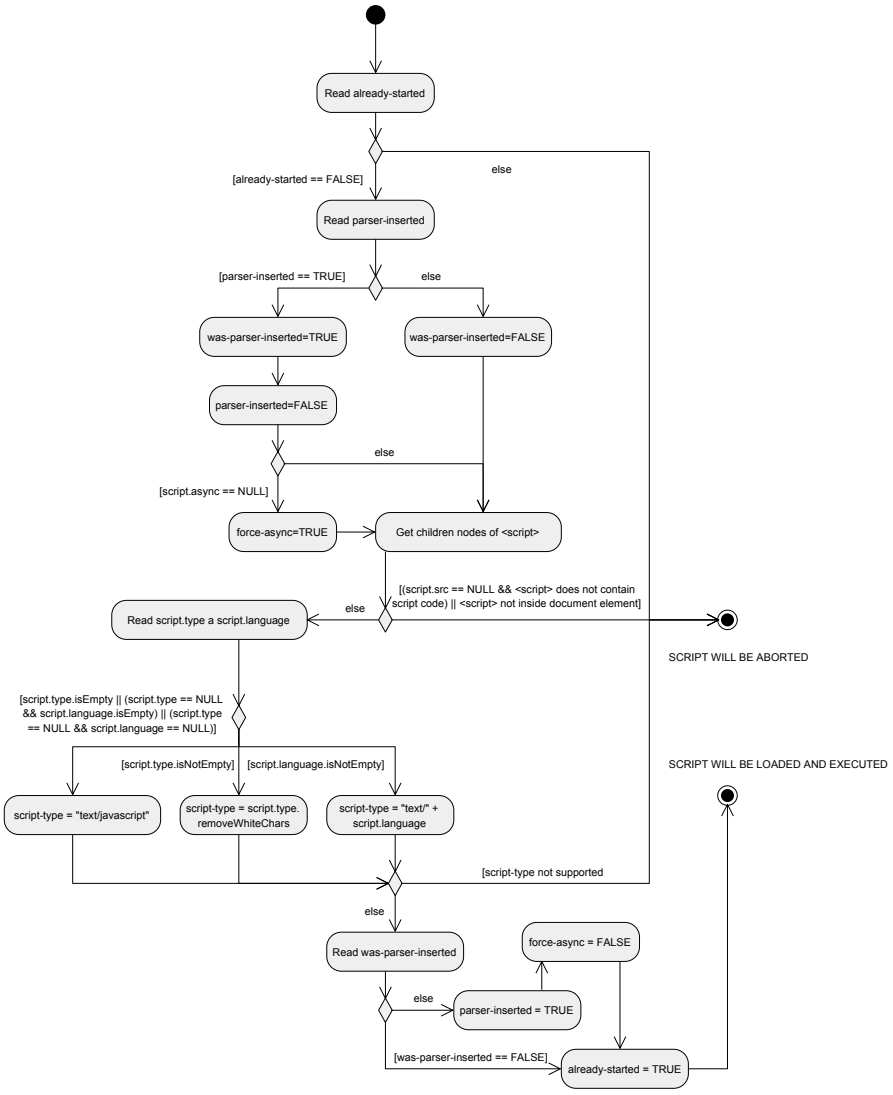
Na elektronický nosič byly přiloženy následující součásti diplomové práce:

- **CSSBox/** – (X)HTML renderovací knihovna;
- **CSSParser/** – parser kaskádových stylů;
- **Doc/** – technická zpráva;
  - **zprava.pdf** – technická zpráva ve formátu PDF;
- **ScriptBox/** – rozšíření skriptování v (X)HTML dokumentech;
  - **demo/** – zdrojové kódy ukázkových HTML stránek;
  - **doc/** – programová dokumentace knihovny v HTML formátu;
  - **src/** – zdrojové kódy knihovny a testů rozšíření;
  - **target/** – výstupní složka pro sestavenou knihovnu;
  - **README** – instalační manuál;
- **SwingBox/** – prohlížeč webových stránek využívající CSSBox.

[1] https://github.com/ITman1/ScriptBox

Příloha B

Předzpracování skriptu



Obrázek B.1: Diagram aktivit znázorňující kroky přípravy skriptu před jeho vykonáním

Příloha C

Manuál

Knihovnu rozšíření lze přeložit více způsoby. Nejběžnější dva způsoby uvedeme v této příloze.

Pokud máme k dispozici přiložené DVD k této práci, lze překlad provést ve vývojovém prostředí Eclipse Kepler s integrovaným nástrojem Maven[1]. K překladu knihovny je zapotřebí mít projekty pracovního adresáře (CSSBox, CSSParser a SwingBox) zkompilované a nainstalované nástrojem Maven (Run -> Run As -> Maven install). Nainstalováním se přidají knihovny těchto projektů do lokálního repozitáře nástroje Maven. Jakmile jsou knihovny nainstalovány, bude možno v Eclipse provést překlad implementovaného rozšíření. Rozšíření je implementováno v projektu ScriptBox.

Nevlastníme-li přiložené DVD a předpokládejme i vývojové prostředí Eclipse, je zapotřebí požadované projekty stáhnout a nainstalovat manuálně. Pro manuální překlad knihovny platí následující předpoklady:

- nainstalované JDK 1.6 nebo novější, kde JDK 1.8 je doporučeno,
- nainstalovaný sestavovací systém Maven,
- vytvořená systémová proměnná `JAVA_HOME` ukazující na lokaci JDK,
- přidaná cesta v systémové proměnné `PATH` k binárním souborům JDK,
- přidaná cesta v systémové proměnné `PATH` k binárním souborům nástroje Maven,
- naklonován repozitář projektu CSSBox [2],
- naklonován repozitář projektu CSSParser[3],
- naklonován repozitář projektu SwingBox[4],
- mít sestavené a nainstalované knihovny projektů CSSBox, CSSParser a SwingBox v lokálním repozitáři nástroje Maven.

Jakmile jsou všechny předpoklady splněny, překlad provedeme příkazem `mvn package`. Během překladu se nástrojem Maven stáhnou také další zbývající závislosti.

[1] http://maven.apache.org/
[2] https://github.com/radkovo/CSSBox
[3] https://github.com/radkovo/jStyleParser
[4] https://github.com/radkovo/SwingBox

Příloha D

Metriky kódu knihovny

Pro získání některých měr bylo využito nástroje Metrics[1].

- **velikost sestavené knihovny** – 526 kB;
- **počet řádků** – 37400;
- **počet řádků (bez komentářů a bílých znaků)** – 19249;
- **počet řádků (uvnitř těl metod)** – 8981;
- **počet balíků** – 45;
- **počet tříd** – 266;
- **počet atributů** – 620;
- **počet metod** – 2200;
- **počet odvozených tříd** – 133;
- **počet přepsaných metod** – 337;
- **hloubka stromu dědičnosti (arit. průměr; směr. odchylka)** – $(2,21; 1,27)$;
- **eferentní propojení balíčků** – $(1,57; 2,80)$;
- **aferentní propojení balíčků** – $(9,53; 10,73)$;
- **nedostatečná soudružnost v metodách** – $(0,21; 0,32)$;
- **McCabe cyklomatická složitost** – $(1,57; 2,00)$;

[1] http://metrics.sourceforge.net/

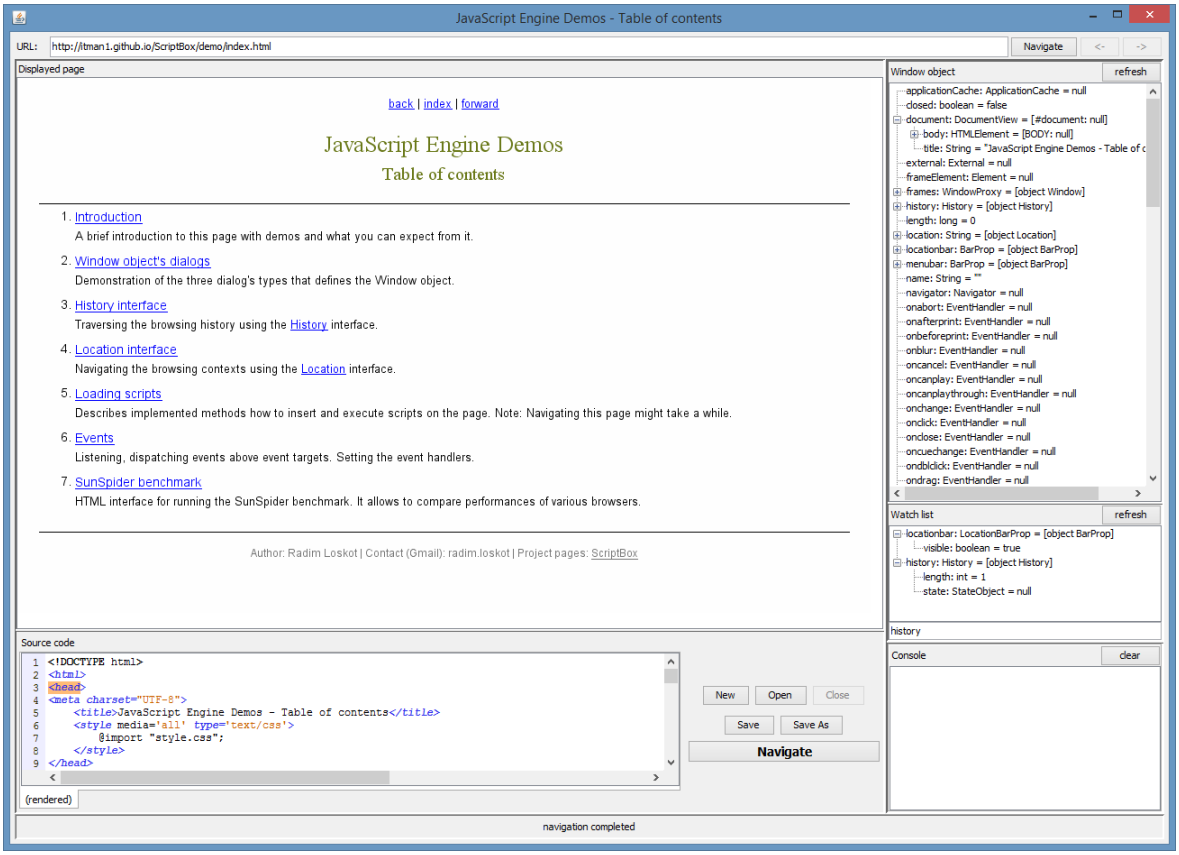
Příloha E

Aplikace jednoduchého prohlížeče

Obrázek E.1: Uživatelské rozhraní jednoduchého prohlížeče

Příloha F

Aplikace testeru JavaScriptu



Obrázek F.1: Uživatelské rozhraní aplikace JavaScriptového testeru

Příloha G

Popis balíků knihovny

Následující seznam obsahuje popis nejdůležitějších balíků projektů. Z cest balíků byla vynechána cesta k hlavního balíku projektu `org/fit/cssbox/scriptbox`.

Výčet nejdůležitějších balíků knihovny:

- **browser/** – jádro uživatelského agenta (uživatelský agent, procházecí jednotka a jednotlivé podporované procházecí kontexty);
- **demo/** – ukázkové aplikace knihovny;

  browser/ – aplikace jednoduchého prohlížeče;

  tester/ – aplikace testeru JavaScriptu;
- **dom/** – třídy objektového modelu DOM (uzly, výjimky, události);

  events/ – třídy reprezentující události DOM;

  adapters/ – adaptéry událostí Xerces na události viditelné ve skriptech;

  script/ – třídy událostí viditelných ve skriptech;

  interfaces/ – rozhraní uzlů dokumentu;
- **events/** – událostní smyčka jádra, plánovače úloh a abstraktní třída úlohy;
- **exceptions/** – výjimky jádra uživatelského agenta;
- **history/** – procházení historie (historie sezení, záznam historie, společná historie sezení a rozhraní `History`);
- **misc/** – pomocné třídy knihovny;
- **navigation/** – navigace mezi dokumenty (navigační kontroler, navigační požadavky a rozhraní `Location`);
- **parser/** – parser dokumentu a úloha zajišťující kompletizaci parsování;

- **reource/** – stahování a zpracovávání obsahu zdroje;

  content/ – registr ovladačů pro zpracování obsahu;

    handlers/ – ovladače pro zpracovávání obsahu;

  fetch/ – registr ovladačů pro stažení zdroje;

    handlers/ – ovladače pro stažení zdroje;

- **script/** – skriptovací architektura;

  adapter/ – registr adaptérů;

  annotation/ – skriptovací anotace pro export objektů;

  exceptions/ – výjimky způsobené jádrem pro skriptování;

  injectors/ – injektory společné pro více skriptovacích enginů;

  javascript/ – implementace skriptovacího enginu JavaScriptu;

    injectors/ – injektory skriptovacího enginu JavaScriptu;

    java/ – implementátor globálního scope JavaScriptu a obaly nativních objektů, atributů a metod;

    wrap/ – obalovací továrny podporované enginem;

  reflect/ – obaly členů tříd Javy – reflexe;

- **security/** – třídy pro bezpečnostní kontroly;

  origins/ – pomocné třídy `Origin` pro určení původu obsahu zdroje;

- **ui/** – rozšíření uživatelského agenta o grafické rozhraní;
- **url/** – třídy pro práci s URL;
- **window/** – implementace globální objektu `Window` a jeho proxy `WindowProxy`.